FAULT- AND YIELD-AWARE ON-CHIP MEMORY DESIGN AND MANAGEMENT

by

Hyunjin Lee

B.S. in Electrical Engineering,

Seoul National University, Korea, 1999

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Hyunjin Lee

It was defended on

April 13th 2011

and approved by

Sangyeun Cho, Ph.D. Associate Professor at Department of Computer Science

Bruce R. Childers, Ph.D., Associate Professor at Department of Computer Science

Rami Melhem, Ph.D., Professor and Dean at Department of Computer Science

Jun Yang, Ph.D., Assistant Professor at Electrical and Computer Engineering

Dissertation Director: Sangyeun Cho, Ph.D. Associate Professor at Department of Computer Science

ABSTRACT

FAULT- AND YIELD-AWARE ON-CHIP MEMORY DESIGN AND MANAGEMENT

Hyunjin Lee, PhD

University of Pittsburgh, 2011

Hyunjin Lee, Ph.D. Ever decreasing device size causes more frequent hard faults, which becomes a serious burden to processor design and yield management. This problem is particularly pronounced in the on-chip memory which consumes up to 70% of a processor's total chip area. Traditional circuit-level techniques, such as redundancy and error correction code, become less effective in error-prevalent environments because of their large area overhead. In this work, we suggest an architectural solution to building reliable on-chip memory in the future processor environment. Our approaches have two parts, a design framework and architectural techniques for on-chip memory structures. Our design framework provides important architectural evaluation metrics such as yield, area, and performance based on low level defects and process variations parameters. Processor architects can quickly evaluate their designs' characteristics in terms of yield, area, and performance. With the framework, we develop architectural yield enhancement solutions for on-chip memory structures including L1 cache, L2 cache and directory memory. Our proposed solutions greatly improve yield with negligible area and performance overhead. Furthermore, we develop a decoupled yield model of compute cores and L2 caches in CMPs, which show that there will be many more L2 caches than compute cores in a chip. We propose efficient utilization techniques for excess caches. Evaluation results show that excess caches significantly improve overall performance of CMPs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 MOTIVATION

At feature sizes below $65nm$, more frequent hard faults due to random defects and process variations pose a serious burden to processor design and yield management [63]. As device size scales downward, a smaller defect size makes it easier to introduce faults that cause critical failures. Process variations also become problematic due to limitations in lithography and process control. The primary effect of process variations is on device length and threshold voltage, which can adversely impact timing and leakage. Amplified process variations require that operational margins widen, making it difficult to construct functioning chips with worst case design.

This problem is particularly pronounced in the memory hierarchy of a processor chip. As the march continues toward large on-chip memories, more of the total transistor budget is devoted to memory. For example, in Intel's Montecito processor [53], the memory arrays for the L2 and L3 caches account for well over 60% of the total chip area. There is a greater likelihood of defects and variations in memory as these structures grow in size because memory transistors are some of the smallest and most timing sensitive. Traditional techniques for masking memory faults use redundancy, where functional spare elements (e.g., columns and/or rows) take the place of defective ones. However, this approach will have to devote a large chip area to the spares in future nanometer-scale technology [2], leading to an adverse interplay with yield.

Vulnerable on-chip memory would be more problematic in chip multiprocessors (CMPs). CMPs integrate multiple cores in a single chip, which require many additional functions not found in single-core processors. One example of additional functions in CMPs is main-

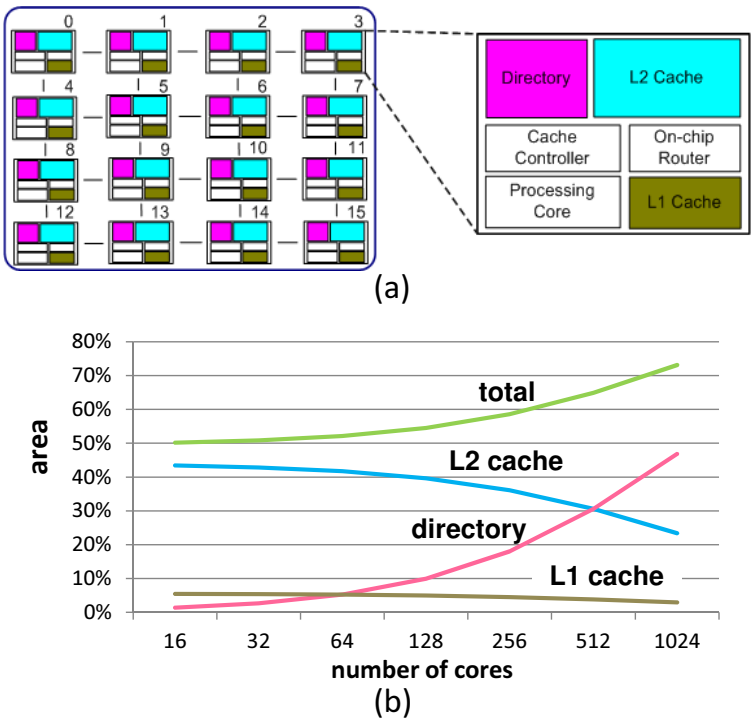


Figure 1: On-chip memory area projection in many-core CMPs.

taining cache coherence. For small-scale CMPs (i.e., core counts<16), broadcasting can be used for cache coherency [40, 38]. However, as the number of cores is expected to grow significantly [11, 40, 5, 65], cache coherency would be maintained by directory-based mechanisms. If the number of cores grows up to hundreds like specialized processors [54, 29], on-chip directory memory would consume a large area. Figure 1 shows 16-cores tile-based CMP (a) and memory area proportion as the number of cores in a chip (b). As we can see in Figure 1(b), the on-chip memory area ratio is increasing with more cores because of the quadratic increase of the directory area. Vulnerable on-chip directory memory may have a severe impact. For example, if a faulty block in on-chip directory memory is masked using the disabling technique, all corresponding L2 and L1 cache blocks should be disabled. Therefore, as the number of cores increases, the potential impact of directory memory faults will correspondingly grow.

Besides salvaging on-chip memory elements, CMPs have more chance to salvage a chip using their inherent core redundancy. When a non-memory functional block or a wide

range on-chip memory area is damaged from faults, single-core processors cannot help being discarded. However, CMPs have many identical cores, which can be disabled with the reduced overall performance [5, 54]. Using core disabling technique, coarse-grain (i.e., bank level) on-chip memory salvaging can be possible if disabled cores have sound on-chip memory banks. While most on-chip memory salvaging techniques uses circuit-level approaches, bank level memory salvaging chance would greatly improve overall performance of CMPs.

While yield-aware on-chip memory design have aforementioned challenges, traditional fault tolerant approaches are either too expensive to utilize or not *efficient* for various memory elements including L1/L2 caches and on-chip directory memory in current and future faulty environments. Traditional techniques, such as redundancy and ECC, impose fault-free functional blocks on the circuit layer so that processor architects can simply assume all functional blocks work correctly. In these design approaches, processor architects assume all microarchitectural functional blocks should work with 100% performance of fault-free functional blocks. Therefore, they either try to mask *all* faults or simply disable unusable memory blocks up to predetermined disabling threshold without considering design specific characteristics for different memory elements.

Architectural yield-aware on-chip memory design, if possible, is expected to have the following merits. First, fault masking requirements for circuit level might be relaxed with the architectural technique supports.This relaxed circuit-level design with the help of architectural techniques may cause negligible performance degradation (e.g., less than 0.1%), while the design cost (i.e., area) might be much less. Second, design specific characteristics, such as coherence protocols for on-chip directory memory, may be utilized to improve yield without sacrificing performance. Third, it may utilize CMPs' characteristic to cooperatively improve yield as well as performance. For example, with abundant compute cores and L2 cache banks in CMPs, architectural techniques may improve overall throughput in the multiprogrammed application environment. Lastly, with small design effort, run-time reliability information may be monitored. OS may support reliability-aware scheduling algorithm, which schedules more threads to more reliable cores.

1.2 OVERVIEW

Processor architects usually consider high level design characteristics such as performance and power consumption at the design time. On the other hand, low-level circuit engineer concerns manufacturing related goals such as yield improvement and cost reduction. Design for manufacturability (DFM) was proposed and widely used by circuit engineers. This technique utilizes more reliable device model or physical placement, which reduce overall fault rate. However, ever increasing fault rates cause DFM to be imperfect or to incur large area overhead. So far, there has been little cooperation between circuit level engineers and processor architects to tackle low yield problem. We believe if processor architects utilize the low level information to design a processor architecture, various dynamic design approaches would be possible. For example, if a function block is more likely to have faults, processor architects add another function block to increase overall yield at the design time. This design decision should be cautiously determined for it is strongly interdependent on many parameters like area, performance, and power.

1.2.1 Approaches

We suggest two distinct design approaches: yield aware design framework and architectural yield enhancement.

First, yield aware design framework should enable architects to predict the impact of potential architectural designs in terms of yield, area, and performance. To provide this goal, the design framework should be able to translate low-level device parameters into high-level architectural fault information. Architects can utilize the high-level knowledge to choose the better architecture design not only for better performance, but also for higher yield and smaller area. Choosing best design choice may vary depending on design goals. For example, architects can choose a slower design with a large gain of yield. These design choice should be cautiously made for benefits of one and losses are strongly interdependent.

Second, based on the given knowledge, on-chip memory elements should be protected with architectural technique as well as circuit level protection mechanism in a cooperative

manner. In this design approach, architectural characteristics of each memory elements can be used to build a customized fault masking techniques for each memory element. For example, on-chip directory memory protection can be achieved using coherence protocol's behavior without incurring large area overhead or performance penalty. Proposed techniques will be evaluated using the developed design framework, compared with other previous techniques.

1.3 THESIS CONTRIBUTIONS

This thesis studies architectural approaches for fault- and yield-aware on-chip memory designs, which deviate from traditional circuit-level on-chip memory yield enhancement techniques. It opens up a new research direction for on-chip memory designs, which utilize circuit layer data to build fault-tolerant microarchitectures. Furthermore, the proposed approaches, with the knowledge of circuit defect data, show architectural on-chip memory design approaches greatly improve overall yield and reliability without sacrificing performance.

Through the study of the proposed approaches, this thesis makes the following contributions:

- A new high-level evaluation methodology for various yield management strategies is proposed. This includes a model that can relate circuit-level defects and process variation in caches to faults at the organizational level. For evaluating different design approaches, a novel metric (YAP) was developed.
- A new yield model for processor components is developed. It is a "decoupled" yield model to accurately calculate the yield of various processor components that have both logic and memory cell arrays. Based on component yield modeling, this thesis performs an availability study for compute cores and low-level cache memory with current and future technology parameters to show that there will likely be more functional caches available than cores in future CMPs.
- A set-remapping graceful degradation scheme is proposed for L1 and L2 caches. It achieved a higher yield than redundancy approaches at negligible performance cost. In

terms of YAP, it is always better than all other traditional schemes including redundancy, ECC, and delete schemes.

- For on-chip coherence directory memory, a novel on-line error detection/correction scheme for hard/intermittent faults and soft error with negligible area overhead is proposed. For hard and intermittent faults, this scheme detects faults by checking a directory entry's integrity on each directory entry read and write event. When data sharing information is found damaged from hard/intermittent faults, it uses a speculative correction strategy to guarantee correct program execution with negligible performance overhead. A single parity bit provides robust soft error detection, even in the presence of multiple hard/intermittent faults.
- Based on the proposed decoupled yield model, this thesis proposes a novel excess cache utilization architecture, StimulusCache, for CMPs. Novel policies and mechanisms are developed for up-to 32-core CMPs. StimulusCache monitors the cache usage of individual threads and limits certain threads from using the excess caches depending on thread benefit from excess cache. StimulusCache is shown to consistently boost the performance of all programs with no performance penalty.

1.4 THESIS ORGANIZATION

The rest of this thesis is organized as follows. Chapter 2 presents background and related work including traditional on-chip memory fault masking techniques and management schemes. Chapter 3 explains the experimental infrastructure which is the foundational research tools for studying fault and yield aware cache design. It also introduces a decoupled yield model, which shows future CMPs have more sound L2 caches than compute cores. Chapter 4 presents two fault and yield aware L1 cache design approaches: graceful degradation and set-remapping. Architectural yield-aware on-chip directory design is presented in Chapter 5. Chapter 6 proposes a yield and performance aware L2 cache design, StimulusCache. Finally, the summary of this work and the future research plan are highlighted in Chapter 7.

2.0 BACKGROUND AND RELATED WORK

In this chapter, we describe previous studies that are related to this thesis. First, we describe faults and errors and their impact on chip yield with technology scaling. Second, we discuss how faults are manifested in L1/L2 caches with a detailed cache memory architecture. Third, on-chip directory memory's architecture for chip multiprocessors are explained. Fourth, we review yield improvement techniques for on-chip memory. Finally, on-chip memory management schemes for chip multiprocessors are discussed.

2.1 FAULTS AND YIELD WITH TECHNOLOGY SCALING

A *fault* is an event and cause of an *error* which can eventually lead to a *system failure*. For instance, a fluctuation in the power supply that is larger than the design margin and an alpha particle hitting the chip circuit are faults. If a circuit state is affected by a fault (e.g., a bit flip in an SRAM cell), an error is said to occur. When the changed circuit state is eventually propagated to the program state, a system failure can happen. Not all errors lead to a system failure, however, since faults and errors may occur to a circuit portion that is not currently in use.

Projections suggest that future microprocessors with advanced nanometer-scale CMOS technology will be subject to three classes of faults: *hard faults*, *intermittent faults*, and *transient faults* [63, 6]. Hard faults reflect irreversible physical damage (defect), caused by imperfect material and process. Intermittent faults happen due to unstable or marginal hardware, activated by changed operating conditions, e.g., higher temperature or lower voltage. Both high temperature and low voltage cause circuit speeds to slow. Transient, soft

errors are caused by charge-assuming particles striking sensitive devices and reversing stored logic states. Cosmic rays and alpha particles generated from chip packaging material are known to cause soft errors.

As circuit technology has entered the sub-65nm regime, much attention has been paid to the impact of process variation and lifetime reliability issues [12]. Process variation refers to the fluctuations in process parameters observed after fabrication. These variations result from a wide range of factors during fabrication which determine the ranges of variations and can lead to designs that deviate significantly from their specification. Furthermore, lifetime reliability issues become more pronounced with technology scaling which increases power densities in the processor. Aging phenomena such as electro-migration, stress migration, gate-oxide breakdown, and negative bias temperature instability (NBTI) can also give rise to hard faults while the chip is operational [67]. Kumar et al. [41] find that NBTI can degrade the read stability of SRAM cells by reducing their static noise margin (SNM) by as much as 9% in 3 years.

2.2 FAULT MANIFESTATIONS IN CACHE MEMORY

When a processor architect considers faults in cache memory, low-level defects and their immediate effect, such as a single bit failure, may not be seen at the architectural level. It is beneficial for an architect to identify faults at the architectural level instead of the low-level especially when evaluating the impact of such faults at an early design stage. This subsection briefly discusses how low-level defects and process variations can manifest themselves at the cache organizational level [47].

Figure 2 depicts a typical set-associative cache for the following discussion. Major components in a cache include memory cells in the tag/data arrays, wires (wordline/bitline/bus), supporting logic (decoders/hit-miss logic) and peripheral circuits (sense amps/drivers).

All major components in a cache, including memory cells, wires, logic, and peripheral circuits, are subject to defects [41, 67]. *A cache fault will occur if a defect in a cache component interferes with any step in a read or write operation.* The critical cache access

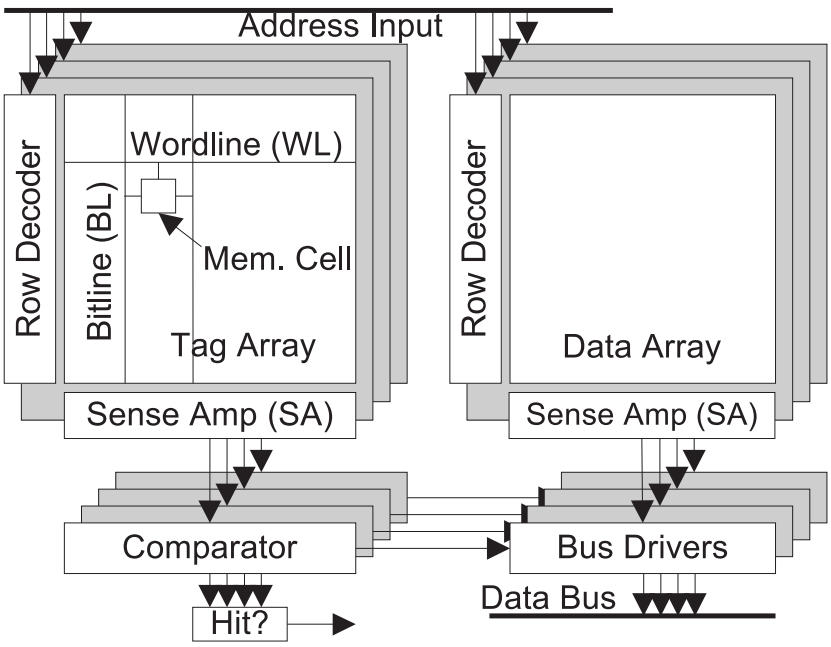


Figure 2: A 4-way set associative cache structure.

path is: ⟨address input, decoding, wordline (data array), memory cell, bitline, sense amp, data bus, data output⟩ and ⟨address input, decoding, wordline (tag array), memory cell, bitline, sense amp, comparator, hit/miss logic⟩.

Defects in cache can manifest themselves in a number of ways at the cache organizational level. First, *individual cache lines* can become faulty and unusable. For example, if a memory cell in a tag or data array has a defect, the corresponding cache line will be unusable. This cache line fault model is commonly used in previous studies [2, 61, 62]. We note that SRAM read stability is aggravated with scaling and cache lines whose cells suffer from reduced signal-to-noise ratio (SNR) will be more frequent in the future. Second, an *entire cache set* may fail. For example, process misalignment can lead to a large number of unstable memory cells, which may cause the loss of cache sets. When there is a defect in the row decoding logic, specific wordlines stick to ground/V_{DD} or they may float. As a result, an entire cache set becomes unusable. In certain cases, defective memory banks can lead to losing a group of cache sets. Third, an *entire cache way* can become faulty due to marginal bitlines or

degraded sense amplifiers. Lastly, the *entire cache* can be lost as a result of a few critical defects in shared resources. For example, defects in the hit logic, the address bus, or the data bus will interfere with every cache access. Consequently, it becomes impossible to use the cache memory.

2.3 ON-CHIP DIRECTORY MEMORY ARCHITECTURE

As the number of cores in a chip is expected to grow, many ideas from multiprocessor systems research will be brought into chip multiprocessors architecture research. For example, on-chip cache memory management techniques, especially memory coherence, will follow similar approaches of shared memory management in multiprocessor systems. Thus, we briefly summarize memory coherence techniques in multiprocessor systems and introduce on-chip memory architectures as well as the baseline cache coherence protocol.

Many multiprocessor systems implement shared memory and private cache in each processing node to ease programming and enable fast access to memory [26, 43]. These systems use a coherence protocol to guarantee that processors access up-to-date memory data in the presence of data sharing via private caches. In general, there are two types of coherence protocols, *broadcast-based* and *directory-based*. Broadcast-based protocols resort to broadcasting messages for coherence actions using a "snooping" bus while directory-based schemes send one-to-one messages using a coherence directory. Broadcast-based snooping protocols are not scalable to large multiprocessors because coherence messages use up much bus bandwidth when there are many processors sharing data [1]. Directory-based protocols are more scalable due to accurate data sharing information kept in the coherence directory. For instance, when a CPU issues a read request to the shared memory, the coherence directory is consulted first (using a one-to-one message) to identify a CPU that has up-to-date data. On a write request, the directory is consulted and the directory controller, if needed, will forward invalidation messages to specific CPUs that actually have the corresponding cache block. As the number of cores in CMPs increases, scalable directory-based coherence protocols will be used in future CMPs [89, 16, 19]. Figure 2.3(a) shows a CMP architecture that

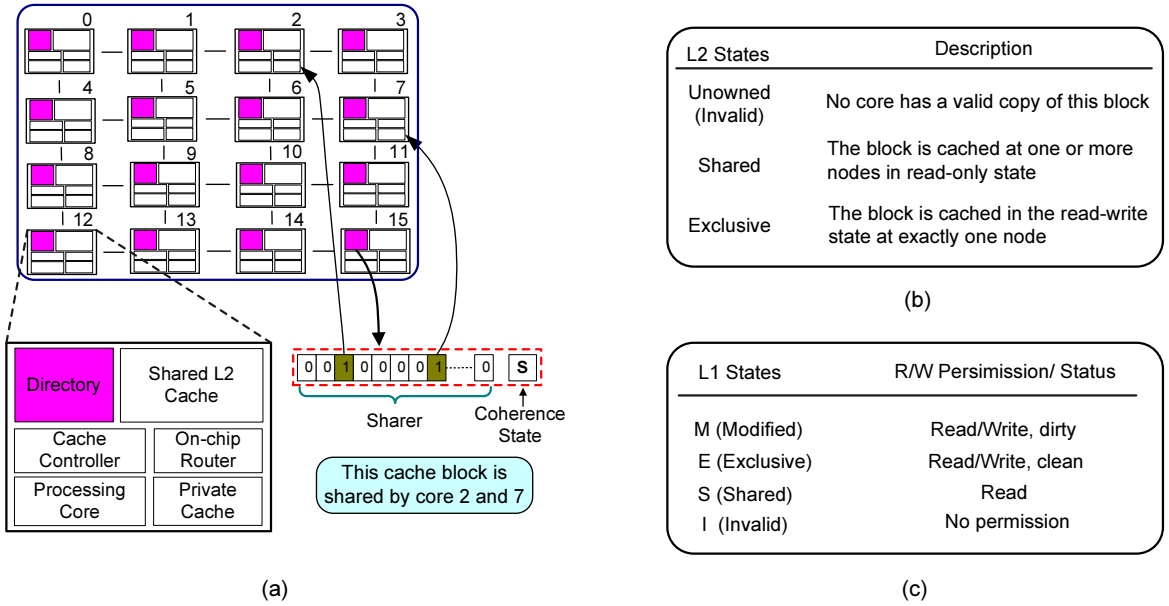


Figure 3: (a) A 16-core CMP architecture with shared L2 cache. Both coherence directory and the shared L2 cache are distributed. Each directory entry, associated with a L2 cache block, has a sharer field and a state field. The directory entry example in the figure shows that the corresponding cache block is shared by two cores, core 2 and 7. (b) and (c) are SGI Origin2000 [43] protocol examples: (b) L2 cache block states and their description. (c) L1 cache block states and their description.

uses a directory-based cache coherence protocol. Each core in the figure has a portion of the directory memory and the shared L2 cache.

The coherence directory is comprised of a series of entries that record coherence-related information for associated cache blocks. Each entry has two fields, *sharer* and *state*. The sharer field shows which cores have copies of a shared data block. The sharer field is typically a bit vector (one bit per core) whose width is the number of cores in the system. Hence, given N cores, the sharer field would hold N bits.

The state field keeps the coherence state of the block. While different cache protocols may use a different set of states, there are five states that appear commonly in the literature [78, 1, 36]: **I** (*Invalid*), **S** (*Shared*), **E** (*Exclusive*), **M** (*Modified*), and **O** (*Owned*). The cache block state information is kept ar the directory and at each core's private cache. As an example, Figure 2.3(b) and (c) list the coherence states a L2 and L1 cache block can have, respectively, in the SGI Origin2000 multiprocessor system [43]. Because the size of the state field does not increase with the number of nodes, we note that the sharer field will dominate

the directory memory area in future large-scale CMPs (i.e., $N > 16$).

In the directory entry example of Figure 2.3(a), core 2 and 7 have a private copy of a shared cache block in the L2 cache of core 15. The corresponding directory entry for the block in core 15 has the Shared state and its sharer shows that core 2 and 7 have cached the block in their L1 cache. The coherence state of the same cache block in core 2 and 7 is Shared, (i.e., read only). When core 2 and 7 read the block, they do not need to take coherence actions. However, when either core intends to update the block, it must acquire the exclusive privilege for the block first by coordinating with the cache coherence controller. The controller then sends appropriate coherence messages to cores that have the cache block to avoid potential stale data accesses, according to the coherence protocol.

Memory accesses from cores may or may not entail coherence actions. For instance, when a core reads from a valid cache block that is already in its private cache (cache read hit), no coherence actions are needed. However, when a core intends to update a shared cache block or access a block that is not currently in its private cache, coherence actions are incurred. In the former case, the required coherence actions include invalidating all current sharers of the cache block (using the information in the corresponding directory entry) and updating the directory entry with the new cache block state and owner. In the latter case, the missing cache block will be brought to the core's private cache (the directory is checked for another processor to provide the data) and the corresponding directory entry will be updated.

Depending on a block's sharers, the sharer field and state field for that block may be updated together or separately. When a cache block is newly brought to the on-chip memory hierarchy, a corresponding directory entry will be allocated and its content, both the sharer field and the state field, updated. When a new core reads from an existing shared cache block, the block's sharer field will be updated, but the state field remains unchanged. In another example, when a core writes to a shared cache block, both the sharer field and the state field of the directory entry need be updated.

As we showed previously, maintaining the data sharing information in the directory memory accurately is necessary for correct program operation.

2.4 ON-CHIP MEMORY SALVAGING TECHNIQUES

In this subsection, we describe three conventional fault covering schemes for memory structures, which could be used for the directory memory as well: *circuit-level redundancy*, *error detection and correction codes*, and *graceful degradation*.

Circuit-level redundancy. Redundant rows, columns, and sub-arrays have been used to tolerate hard errors at the time of manufacture. When faulty bits are detected, damaged blocks with faulty bits are replaced by sound redundant blocks, i.e., rows, columns, or sub-arrays [13, 21, 49]. Moreover, with help from built-in self-test (BIST) circuitry, mapping of circuit-level redundancy can be done at power-on time, rather than manufacture time [32]. When there are more faulty blocks than can be covered by available redundancy, however, two choices remain: Discarding the chip, which causes yield loss, or salvaging remaining memory capacity by removing faulty memory elements from usage. The latter approach, graceful degradation, is discussed below.

Circuit-level redundancy is considered inefficient. It is not scalable because of large area overhead reserved during design time. Recovering from a single bit fault requires replacing a whole row or column. Because a chip built with future technology (e.g., 32nm) is expected to have many more faults [12], guaranteeing a target yield using circuit-level redundancy will inevitably lead to a large chip area overhead.

Error detection and correction codes (EDC/ECC). Single error detection codes (SED) such as parity and single error correction double error detection codes (SECDED) are widely used in memory designs [7, 40, 70]. Although multi-bit error correction codes are available (e.g., double error correction), they are often not suitable for use in time-critical memory due to their computational complexity and high power consumption. SECDED requires $m + 2$ code bits for 2^m data bits. A popular configuration of SECDED is 8-bit code for 64-bit data word (i.e., $m = 6$). ECC is mostly employed for soft error protection in recent processors [24, 32, 70]. However, ECC can cover up to the number of errors, determined by its correction limit (e.g., 1 bit for SECDED, 2 bits for DECTED) regardless of the error sources. With ECC's versatility, there have been attempts to use ECC to cover hard faults [2, 76]. The main trade-off in this case is enhanced yield (hard fault is covered) versus sacrificed

immunity against other types of faults such as wear-out fault and soft error. If the single error correction capability of SECDED is used to cover a hard fault in a cache block, for example, the corresponding block becomes vulnerable to a soft error.

A two-dimensional error coding was proposed to guarantee soft error immunity when a cache block is damaged by a hard fault [39]. Although such a smart hardware scheme increases yield and lifetime reliability considerably, multi-bit correction in this scheme usually requires a huge cycle penalty (e.g., complete test of memory using BIST). Multi-bit error correction schemes with small cycle penalty have been proposed [51, 8]. However, these schemes' large area cost makes them inadequate for the directory memory. As shown in Section 6.3, redundancy and ECC can be used in a synergistic way to increase yield [77]. However, if hard errors in any blocks are covered by ECC, ECC cannot protect these blocks from soft errors.

Graceful degradation schemes. Due to the deficiencies of other approaches, researchers have considered graceful degradation strategies as a viable approach to covering faults in various memory structures [2, 17, 32, 42, 47, 48, 61, 62, 85]. The basic idea of graceful degradation is, when applied to cache memory, to "delete" a faulty cache portion so that the processor will not use the cache blocks that fall into the faulty portion. Because the loss of a few cache blocks will degrade processor performance to only a limited extent, a graceful degradation approach is more flexible and cost-effective than redundancy approaches [61, 62, 47, 48]. Recently, Intel introduced a graceful degradation mechanism for low-level cache memory, dubbed Cache Safe Technology [17]. It disables a cache block when it detects that an intermittent fault in the cache block transitions to a hard fault, based on observed ECC actions. The maximum number of disabled blocks is limited to 32 to minimize the added hardware resources and performance penalty. A similar technique has been adopted in the POWER5 [65] and POWER6 processors [32]. There is programmable steering logic that is initiated by BIST circuitry and is activated during processor initialization to replace faulty bits. When the L3 cache memory detects faults that exceed a pre-determined threshold, the processor invokes a dynamic L3 cache line delete action.

3.0 EXPERIMENTAL INFRASTRUCTURE FOR STUDYING FAULT AND YIELD AWARE CACHE DESIGN

The design of critical processor components like cache memory at the architectural level is complicated by the requirement of meeting a number of important, yet potentially conflicting design goals. Today, while performance has been the key design goal in high-performance processor designs, the cost of a design, typically measured in chip area, remains a serious target to optimize for the competitiveness of the final design. Yield is yet another important design consideration that affects the cost structure of the product. A processor architect has to make judicious trade-offs between these interacting design goals–performance, area, and yield.

Evaluating a cache design in terms of different design goals can be tricky. Ideally, how all the design goals are met by a specific design should be evaluated simultaneously and if possible, early. This is especially desirable if an architecturally visible yield-enhancement scheme such as set remapping, presented in Section 4.2, is adopted because it impacts performance, area, and yield at the same time. In reality, however, not all necessary information is typically available early in the design cycle. Relating one aspect of a design (e.g., yield, mostly determined by manufacturing characteristics) with another (e.g., performance) is not straightforward.

Because of the interdependency between how one designs a cache memory and how low-level manufacturing characteristics affect its yield, we propose an integrated framework for designing defect-tolerant cache memory that considers hardware defects, cache yield, fault-masking schemes, cache area, and cache performance simultaneously. In the proposed framework, we first derive high-level fault manifestations at the organizational level from low-level defect specification. As a result, fault-aware performance simulation can be done

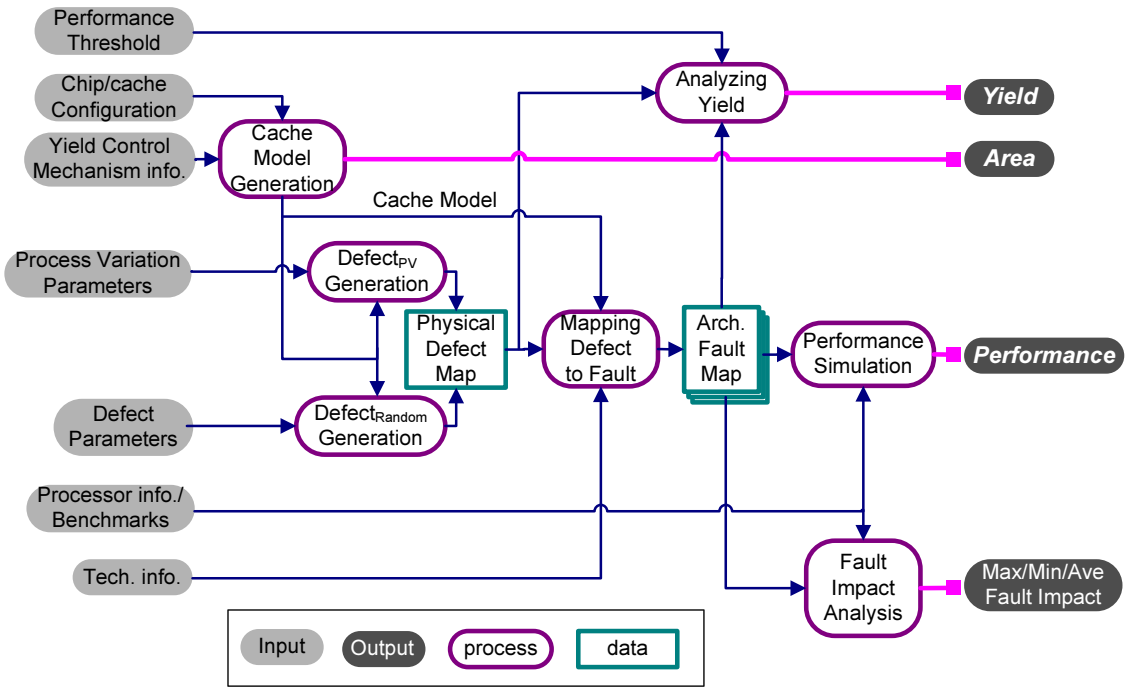


Figure 4: DEFCAM design flow overview.

early with the derived fault information. We then evaluate a cache design in terms of yield, area, and performance efficiently in an early design stage so that important cache design decisions can be made in an informed manner. We call our framework DEFCAM (A Design and Evaluation Framework for defect-tolerant CAche Memories).

3.1 INTEGRATED DESIGN FLOW

DEFCAM's goal is to provide a processor architect with an integrated framework to evaluate a cache design's yield, area, and performance.[1] Figure 4 shows the DEFCAM framework. The design flow can be easily extended to include other traditional metrics, such as power and energy consumption. However, power is not the focus of this thesis because good power

[1] Yield usually refers to the proportion of dies on a wafer that perform properly up to a design specification. In this paper, we extend the use of "yield" to a cache within a processor chip.

estimation tools already exist, which can readily be integrated into the framework [80, 87].

There are three obstacles in current methodologies that have to be addressed to achieve our goal. First, during early design exploration, a processor architect needs to understand how defects and process variations affect the cache to select a good yield management scheme. There are complex trade-offs between yield, area, and performance, especially when yield control mechanisms with different area and defect coverage are considered [47, 81]. Second, defects and process variations manifest themselves at the physical level and need to be related to the cache microarchitecture under consideration. However, the physical design is unknown during early design exploration. Finally, because a particular yield management scheme can affect several design layers from the application level to the physical level, efforts from independent design groups have to be integrated.

Our design flow addresses these problems in the following way. From a "cache microarchitecture specification," the design flow automatically derives a "physical cache model." The physical cache model approximates the anticipated implementation of the given cache specification. For instance, the number of wordline segments (in Table 1) can be used to calculate the number of row decoders in the physical cache model. Defects and process variations are automatically placed in the physical model according to a "defect model." The defect model is based on the characteristics of the target technology. The defects and process variations at the circuit level are then mapped to the organizational level to determine how they affect the cache (e.g., which cache lines failed). Because some yield management schemes do not guarantee program performance, our methodology simulates—at the architecture behavioral level—a cache with a set of given faults. The simulation is done automatically only in cases when the yield management scheme can impact program performance (e.g., when the number of faults is bigger than the number of redundant rows). To ensure statistically valid coverage of many possible fault manifestations, the methodology uses extensive sampling at the virtual wafer level to derive and evaluate different defect and process variation scenarios (e.g., 100 wafers per each scenario).

To use the design flow in Figure 4, several inputs have to be specified. Table 1 lists these inputs. The processor architect would typically specify the cache organization, program workload and processor architecture. Information about the target technology can be

provided by circuit and process engineers. The output of the design flow includes yield, area, and program performance. In our current implementation, obtaining the output from the input is fully automated.

Table 1: Input parameters.

Cache configuration	Description
A	Set associativity
B	Block size (in bytes)
C	Cache size (in bytes)
N_{dwl}	Number of wordline segments
N_{dbl}	Number of bitline segments
N_{spd}	Number of sets mapped to a single wordline
Yield control mechanism	**Description**
Redundancy	Number of redundant rows
ECC	Number of correctable bits
Disabling	Line/set/way disabling
Set remapping	Number of target set candidates
Defect	**Description**
Density	Number of defects in unit area ($1mm^2$)
Clustering	Average number of clustered defects
Size	Average size of defects
Process variation	**Description**
Inter-die variation	$\Delta V_{th-inter}$ (in V)
Intra-die variation	$\Delta V_{th-intra}$ (in V)
Technology information	**Description**
Wafer size	Diameter (mm)
Threshold voltage	V_{th} (in V)
cx, cy	SRAM cell dimension (in nm)
Processor	**Description**
Pipeline	In-order/out-of-order, issue width, number of ROBs
Cache	L1/L2 cache architectural configurations
Branch predictor	Number of entry, bi-mod/combined/g-share
Memory	Access latency, bus width

3.2 MODELING DEFECTS

Procedures for generating and projecting defects into physical layer were developed. We use a $300mm$ silicon wafer model and the die area model from the specified processor model.

A hard defect is a spot defect on the physical layout of the wafer. Based on the physical information input to DEFCAM, we randomly generate physical defects with defect density and clustering effect factors. One can also set defect sizes from the ITRS projection data [35]. After generating defects, we inject them into a wafer according to the location of defects in the wafer. We use a Gaussian random distribution model for the defect locations and sizes in this work, but any distribution model can be specified. Clustering factors can represent the nature of the real processor's defects [25]. Two defect generation blocks (i.e., physical defect generation and process variation generation) and the physical defect map in Figure 4 illustrate these processes in the DEFCAM design flow. After generating physical defects, we translate low-level defects into architectural level faults using the methodology described next.

3.3 TRANSLATING DEFECTS INTO ARCHITECTURAL FAULTS

To accurately translate physical defects into architecturally visible faults, a detailed cache specification is necessary. The most important cache design parameters are architectural parameters, organizational parameters (e.g., banking), and physical layout information such as array geometry and SRAM cell layout. Our design flow provides an interface to describe a cache's geometry inside a chip.

Architectural parameters are a 3-tuple: A (associativity), B (block size), and C (cache size). We consider three organizational parameters: N_{dwl}, N_{dbl}, and N_{spd}, which determine the internal sub-banking by specifying how wordlines and bitlines are partitioned [80, 83]. They are listed as cache configuration parameters in Table 1. They also define cache line to sub-bank mapping. Cache geometry is modeled as a hierarchy of non-overlapping rectangles, from the wafer and chip down to cache and cache components. Cache components include memory arrays and control logic. To ease illustration, we use the derived parameters in Table 2. These parameters are used for calculating faulty way/set numbers (i.e., architectural fault information) from defect parameters (i.e., circuit-level defects or process variations). For the sake of presentation, we consider only the data array in the following discussion.

Table 2: Derived parameters for mapping defects to faults.

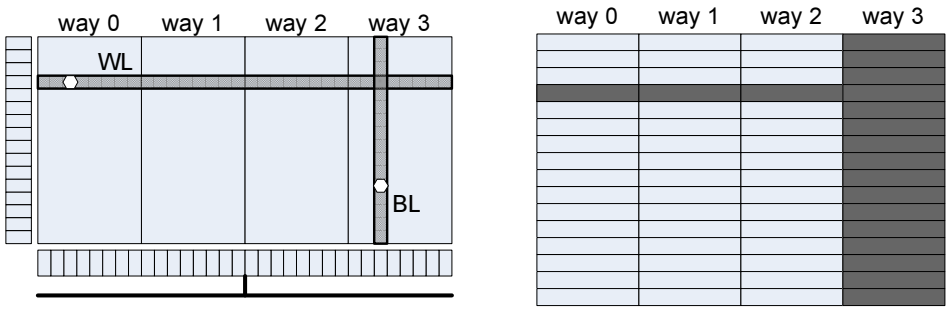
Parameter	Description
px, py	Coord. of a given physical defect ϕ from physical defect map (in nm)
W_b	Block width; $8 \cdot cx \cdot B$ (in nm)
W_{sb}	Sub-bank width; $A \cdot W_b \cdot N_{spd}/N_{dwl}$ (in nm)
W_{ssb}	Set width in a sub-bank ($N_{spd} \geq 1$); W_{sb}/N_{spd}
px_{sb}	Rel. X-coord. of ϕ in a sub-bank; $px \ mod \ W_{sb}$
N_{set}	Number of sets; $C/(A \cdot B)$
N_{sbl}	Number of sets per bitline; N_{set}/N_{dbl}
N_{pr}	Index of memory row having ϕ; $\lfloor py/cy \rfloor$
N_{xsb}	X-index of sub-bank having ϕ; $\lfloor px/W_{sb} \rfloor$

Our techniques apply equally well to the tag array.

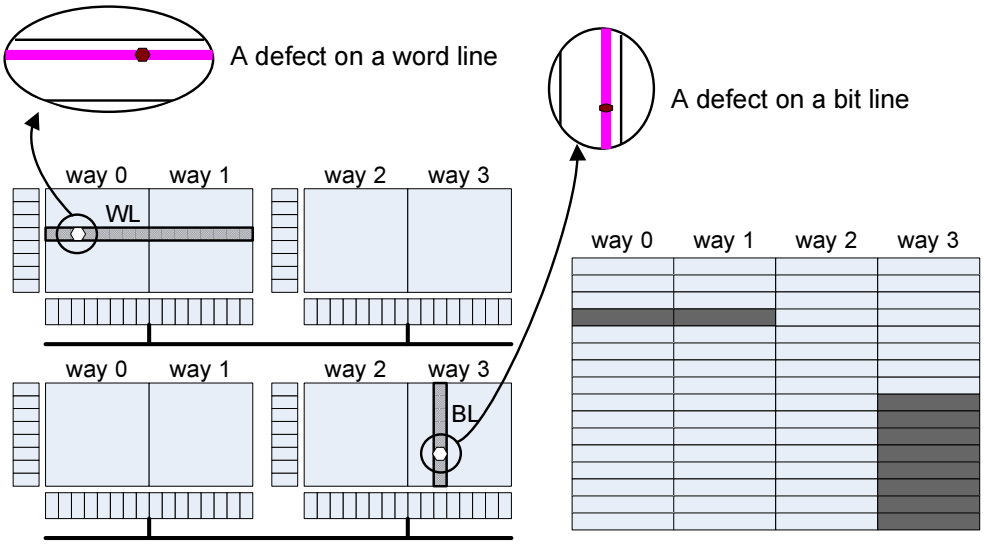
A defect or process variation in the physical model is mapped to the cache microarchitecture based on the affected component. In this mapping, our methodology considers three component types: bitline, wordline, and memory cell. Bitline and wordline faults include sense amplifier faults and row decoder faults, respectively. Once lower level defects are converted to architectural faults, different defects are indistinguishable at the organizational level.

3.3.1 Bitline and sense amplifier

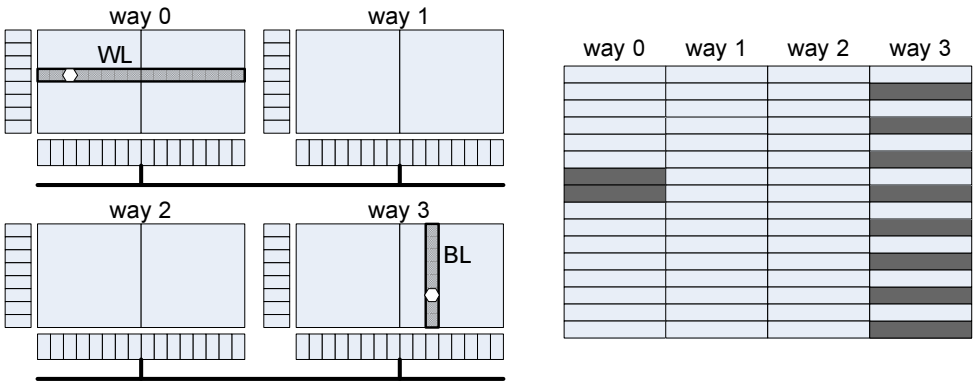
A defect in a bitline or a sense amplifier causes faulty cache lines. Depending on the cache organization, all the lines in a way can be lost as in Figure 5(a) or only a portion of them as shown in Figure 5(b) and (c). The left part of the figure shows some defects placed in the cache at the circuit level, and the right part architectural faults according to the defects. We consider two cases depending on the value of N_{spd}. When $N_{spd} \geq 1$, total $(N_{set}/(N_{spd} \cdot N_{dbl}))$ lines are affected. Within a selected cache way (*way num* in below equations), one cache line in every N_{spd} consecutive cache lines is faulty. In other words, given a defect $\phi = (px, py)$ which shows two dimensional location information, all the sets with their number equal to $(initial\ set\ num + n \cdot N_{spd}\ mod\ (N_{set}/N_{dbl})) \geq 0$, are faulty. *way num* and *initial set num* are



(a) A=4, N_{dwl}=1, N_{dbl}=1, N_{spd}=1



(b) A=4, N_{dwl}=2, N_{dbl}=2, N_{spd}=0.25



(c) A=4, N_{dwl}=4, N_{dbl}=1, N_{spd}=2

Figure 5: Defective wordline or bitline (left) results in various architectural faults (right).

computed as:

$$\begin{aligned} \textit{way num} &= N_{xsb} \cdot W_{ssb}/W_{sb} + \lfloor (px_{sb} \bmod W_{ssb})/W_b \rfloor \\ \textit{initial set num} &= N_{pr} \cdot N_{spd} + \lfloor px_{sb}/W_{ssb} \rfloor \end{aligned}$$

When $N_{spd} < 1$, total (N_{set}/N_{dbl}) consecutive cache lines become faulty. They are identified with:

$$\begin{aligned} \textit{way num} &= \lfloor px/(W_b \cdot N_{spd}) \rfloor \\ \textit{set num, start} &= \lfloor (N_{pr} \cdot N_{spd})/N_{sbl} \rfloor \cdot N_{sbl} \end{aligned}$$

3.3.2 Wordline and row decoder

Like a defective bitline, a defective wordline or a defective row decoder (i.e., an unusable decoder line) can cause unavailable cache lines in a number of different cache ways. All lines in a set can be lost as in Figure 5(a), a few cache lines in a set may become unavailable as in Figure 5(b), or cache lines in multiple cache sets can become faulty as in Figure 5(c). When $N_{spd} \geq 1$, faulty cache lines are clustered in a rectangle (N_{spd} by A/N_{dwl}) whose origin can be computed as follows:

$$\begin{aligned} \textit{set num, start} &= N_{pr} \cdot N_{spd} \\ \textit{line num, start} &= N_{xsb} \cdot A/N_{dwl} \end{aligned}$$

When $N_{spd} < 1$, (A/N_{dwl}) consecutive lines become faulty within a single set. The initial line is identified with the following:

$$\begin{aligned} \textit{set num} &= \lfloor N_{pr} \cdot N_{spd} \rfloor \\ \textit{line num, start} &= \lfloor \frac{\lfloor px/(W_b \cdot N_{spd}) \rfloor}{A/N_{dwl}} \rfloor \cdot \frac{A}{N_{dwl}} \end{aligned}$$

3.3.3 Memory cell

A faulty cache line due to a defective memory cell is determined by the following equations. Again, we consider two cases based on the value of N_{spd}. When $N_{spd} \geq 1$, calculating the set and line number is identical to finding the initial set and way number of the bitline defect case.

$$\begin{aligned} set\ num &= N_{pr} \cdot N_{spd} + \lfloor px_{sb}/W_{ssb} \rfloor \\ line\ num &= N_{xsb} \cdot W_{ssb}/W_b + \lfloor (px_{sb}\ mod\ W_{ssb})/W_b \rfloor \end{aligned}$$

When $N_{spd} < 1$, this is also very similar to the bitline defect case.

$$\begin{aligned} set\ num &= \lfloor N_{pr} \cdot N_{spd} \rfloor \\ line\ num &= \lfloor px/(W_b \cdot N_{spd}) \rfloor \end{aligned}$$

Besides random defects, we also consider the impact of process variations on memory cell reliability. Failures caused by process variations are heavily dependent on operating voltage, frequency, and temperature. Since these conditions are varying factors, the worst case operation condition (e.g., low voltage and high temperature) should be considered at manufacture time. Thus, we assume the worst operating condition as a fixed design point. While there are systematic variations and random variations, we only consider random variations in this study. Process variations are caused by imperfect control over the channel length, width, oxide thickness, and placement of dopants. Among them, random dopants are more important than the others because it causes a threshold voltage mismatch even between nearby transistors [79]. Different threshold voltages vary the transistor's response time and lead to SRAM cell's access time failures and read and/or write failure. These three failures (access, read, and write time failures) compose the probability of failure of one cell. This problem is magnified when adjacent transistors in one SRAM cell have different threshold voltages [2]. The probability of failure increases as the variation of threshold voltages grows. For example, effective probability of failure becomes 1×10^{-3} in $45nm$ technology where ΔV_{th} is $30mV$ [2]. Based on these observations, we consider both inter-die variation ($\Delta V_{th-inter}$, common to devices in a die) and intra-die variation ($\Delta V_{th-intra}$) using a

Gaussian distribution whose standard deviation is given by the BPTM model [10]. We then assign ($\Delta V_{th-inter} + \Delta V_{th-intra}$) to each memory cell. Finally, we determine that a memory cell under consideration has a fault if its ($V_{th} + \Delta V_{th}$) is not within a user-defined range. In essence, whether a memory cell has a fault due to process variations or not is decided probabilistically, similar to [2].

3.4 YAP: YIELD, AREA, AND PERFORMANCE

The yield of a cache memory depends on its area and organization as well as defect distribution and process variation effects. A cache memory with a defect which hinders its normal operation is considered a failed component, unless a yield enhancement technique masks the effect of the defect. The area of a cache depends on its baseline design (specified by design parameters) and the yield enhancing scheme employed. The performance of a cache memory is primarily determined by its architectural parameters. Note that these metrics interplay. For example, a larger cache design may lead to lower yield. If a disabling scheme is used, the resulting yield may be high, but the average performance obtainable from the degraded cache may be low. Therefore, these metrics should be made available together to a designer at an early design phase so that good design decisions can be made.

Our design flow reports these metrics together in a 3-tuple (Y (yield), A (area), P (performance)). To conveniently compare multiple design points with a single number, the three terms can be combined in various ways. For example, when Y, A, and P are normalized to a baseline cache design with no yield enhancement, $Y^l \cdot A^{-m} \cdot P^n$ gives a single number which ranges between 0 and 1. The negative exponent for A comes from the fact that a smaller area cache presents a better design with the same performance and yield. The choice of l, m, and n depends on the emphasis of analysis. If a processor architect is more concerned with area, m should be bigger than other two numbers. $Y^{0.7} \cdot A^{-0.9} \cdot P^{0.4}$ is an example, which stresses area rather than performance.

3.4.1 Area calculation

DEFCAM uses CACTI [80] to calculate the baseline cache area for a given technology. Further, there are three elements that affect the area: row redundancy, ECC, and an availability bit for graceful degradation. For redundant rows, data area grows proportionally to the number of redundant rows. We assume conventional SECDED code (i.e., 8-bit ECC code for 64-bit data) to evaluate the area overhead from ECC. We consider one bit overhead for graceful degradation, which comes from each block's availability bit [62].

3.4.2 Yield calculation

DEFCAM computes cache yields by simulating fault occurrences from the information about cache configuration, low level defects, process variations, and yield enhancing techniques employed. Low-level defects are randomly injected to a user-specified number of dies, which eventually become bit line, word line, cell, or peripheral errors based on the cache configuration. Process variations are calculated for individual cells using the methods described in Section 3.3.3. These two types of errors are projected to higher level architectural fault information. With the architectural fault information, the impact of different yield enhancing techniques is evaluated. If a given yield enhancing technique is unable to cover all faults in a die (e.g., more faults than the number of redundant rows), yield is sacrificed.

3.4.3 Performance simulation

DEFCAM simulates workloads to measure the performance for different architectural fault maps as shown in Figure 4. In our current implementation, the performance simulator is based on SimpleScalar [9] with an additional module to simulate the effects of cache faults with fault masking strategies—delete schemes and remapping schemes—which are described in Section 4.1. This performance simulator enables the user to analyze how much performance is impacted from faults. The performance result also becomes an input to the Yield Analyzer (see Figure 4). When a disabling or remapping scheme is used, a cache achieving performance equivalent to a user-specified threshold or better is regarded as a

sound cache. In this paper, we use either a 95% or 99% performance threshold for the caches using a graceful degradation scheme. For the 95% performance threshold, if any benchmark's performance on a processor show more than 5% degradation from disabling faulty blocks in the cache, it is discarded.

3.5 EVALUATION: A CASE STUDY

To investigate the utility of DEFCAM, we did a case study to select a yield-effective cache design (with YAP) from a set of candidate designs. This study illustrates how a processor architect might use DEFCAM to evaluate several process/defect tolerant cache designs to select one. The first part in this section explains how we specify the design space for the case study, and the second part describes the evaluation of candidate designs.

3.5.1 Specification

The case study is a system-on-a-chip (SoC) with an ARM processor in Table 5. To simplify the study, we assume that defects occur only in the caches. The internal cache organization is derived with CACTI 4.1 [80].

The case study considers four yield management schemes for the cache: no redundancy (baseline), 12.5% and 25% row redundancy, line delete, and static set remapping (from Section 4.2.4). Row redundancy uses spare memory rows to cover defective ones; spares can replace any defective row. The number of spares is the percentage 12.5% or 25% of total cache lines. Line delete uses graceful degradation to permanently disable defective lines. A spare line is not used in place of a defective one, and as a result, program addresses that map to defective lines are not cached. Because error correction codes (ECC) can mitigate the effect of process variations [2], we consider designs with and without ECC. For set remapping, as suggested in Section 4.2, we use the static scheme. The pool of target sets has four target set candidates. Therefore, at most four faulty sets can be remapped. Any faulty sets beyond this limit are deleted.

Our defect model uses a uniform distribution to select defect locations and a Gaussian distribution for defect size. Process variation has a single parameter: $\sigma_{V_{th}}$. The model is configured with a tuple (*defect density*, $\sigma_{V_{th}}$), which includes physical defect and process variation informations in one tuple. The study uses the configurations (1, $30mV$), (10, $30mV$), (100, $30mV$), (100, $20mV$), and (100, $40mV$).[2] Defect density 1 and 10 conform ITRS projection data [35]. We aggressively generate the heavy defect density to 100. $\sigma_{V_{th}}$ 30 comes from 45-nm technology's process variation [2]. $\sigma_{V_{th}}$ 20 and 40 are used to evaluate the impact of different parameter variations. Similarly, we evaluated the yield management schemes for three cache organizations with same cache capacity. We used three configurations with the 3-tuple, $(N_{dwl}, N_{dbl}, N_{spd})$ = (1, 4, 0.5), (4, 4, 0.5), (1, 4, 0.125). The study uses 1,000 total die samples.

To measure program performance, the SimpleScalar tool set [9] is used in DEFCAM to simulate an ARM-like in-order pipelined processor. The workload is MiBench [28] with the large data sets.

Using DEFCAM, we evaluated the yield management schemes. The results are in Tables 3 (in Page 28). In the tables, yield is the ratio of operational caches to total caches. Area is the cache area relative to the baseline. Performance is an average of the benchmarks. In this case study, we use four YAP metrics, labeled M1–M4, denoting $Y \cdot A^{-1} \cdot P$, $Y \cdot A^{-1} \cdot P^2$, $Y^2 \cdot A^{-1} \cdot P$, and $Y \cdot A^{-2} \cdot P$, respectively. As we describe in Section 3.4, power terms of YAP can be defined by the user depending on design goals. There are twelve designs with four cases for line delete and two cases for set remapping. 95% line delete and set remapping discards all caches that have more than a 5% performance degradation. Similarly, 99% line delete and set remapping discards caches above a 1% degradation.

[2] We scaled the defect densities for intuitive presentation and comparison. The defect density 1 corresponds to $0.5/mm^2$.

Table 3: Yield, Area, Performance, and four YAP metrics (M1–M4) in % for different: (a) defect densities, (b) process variations, and (c) organizations. The YAP metrics M1–M4 are $Y \cdot A^{-1} \cdot P$, $Y \cdot A^{-1} \cdot P^2$, $Y^2 \cdot A^{-1} \cdot P$, and $Y \cdot A^{-2} \cdot P$, respectively. Numbers in bold face represent the best YAP values.

(a) (defect, $\sigma_{V_{th}}$)	(1, 30mV)							(10, 30mV)							(100, 30mV)						
Design	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4
No redundancy	83	100	100	83	83	69	83	80	100	100	80	80	64	80	51	100	100	51	51	26	51
No redun. ECC	100	104	100	96	96	96	92	98	104	100	94	94	92	90	68	104	100	66	66	45	63
12.5% redundancy	91	108	100	84	84	77	78	90	108	100	84	84	75	77	69	108	100	64	64	44	59
25% redundancy	94	116	100	81	81	76	70	94	116	100	83	83	77	73	82	116	100	72	72	59	64
12.5% redun. ECC	100	113	100	88	88	88	78	100	113	100	86	86	86	74	95	113	100	82	82	78	71
25% redun. ECC	100	121	100	83	83	83	68	100	121	100	82	82	82	68	95	121	100	79	79	75	65
95% delete	100	102	100	**98**	**98**	**98**	**96**	99	102	100	**97**	**97**	96	**95**	86	102	98	83	81	71	81
99% delete	99	102	100	97	97	96	95	98	102	100	96	96	94	94	82	102	100	80	80	66	79
95% set remap	100	103	100	97	97	97	94	100	103	100	**97**	**97**	**97**	94	98	103	100	**95**	**95**	**93**	**92**
99% set remap	100	103	100	97	97	97	94	100	103	100	**97**	**97**	**97**	94	93	103	100	90	90	84	88

(b) (defect, $\sigma_{V_{th}}$)	(100, 20mV)							(100, 30mV)							(100, 40mV)						
Design	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4
No redundancy	62	100	100	62	62	39	62	51	100	100	51	51	26	51	28	100	100	28	28	8	28
No redun. ECC	71	104	100	68	68	48	66	68	104	100	66	66	45	63	68	104	100	65	65	44	63
12.5% redundancy	78	108	100	72	72	56	66	69	108	100	64	64	44	59	47	108	100	43	43	20	40
25% redundancy	91	116	100	78	78	71	68	82	116	100	72	72	59	64	53	116	100	47	47	25	42
12.5% redun. ECC	95	113	100	84	84	80	74	95	113	100	82	82	78	71	92	113	100	79	79	72	68
25% redun. ECC	97	121	100	80	80	77	66	95	121	100	79	79	75	65	95	121	100	78	78	74	65
95% delete	90	102	97	88	88	79	87	84	102	98	82	82	69	81	83	102	97	80	78	66	78
99% delete	88	102	100	86	86	76	85	82	102	100	80	80	66	79	81	102	100	79	79	64	78
95% set remap	99	106	100	**95**	**94**	**94**	**92**	98	106	100	**95**	**95**	**93**	**92**	98	106	100	**94**	**93**	**92**	**91**
99% set remap	94	106	100	91	91	86	89	93	106	100	90	90	84	88	93	106	100	90	90	84	88

(c) ($N_{dwl}, N_{dbl}, N_{spd}$)	(1, 4, 0.5)							(4, 4, 0.5)							(1, 4, 0.125)						
Design	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4	Y	A	P	M1	M2	M3	M4
No redundancy	55	99	100	55	55	30	55	51	100	100	51	51	26	51	51	100	100	51	51	26	51
No redun. ECC	71	103	100	68	68	48	65	69	104	100	66	66	46	64	68	104	100	66	66	45	63
12.5% redundancy	69	107	100	64	64	44	59	68	108	100	63	63	42	58	69	108	100	64	64	44	59
25% redundancy	81	115	100	72	72	58	64	81	116	100	70	70	57	60	82	116	100	72	72	59	64
12.5% redun. ECC	96	112	100	82	82	79	71	94	113	100	83	83	78	74	95	113	100	82	82	78	71
25% redun. ECC	95	120	100	78	78	74	65	95	121	100	78	78	74	65	95	121	100	79	79	75	65
95% delete	87	101	97	84	82	73	82	87	102	98	85	85	74	84	86	102	97	84	84	73	83
99% delete	82	101	100	80	80	66	79	81	102	100	79	79	64	78	82	102	100	80	80	66	79
95% set remap	99	103	100	**95**	**94**	**94**	**92**	98	103	100	**95**	**95**	**93**	**92**	98	103	100	**95**	**95**	**93**	**92**
99% set remap	93	103	100	90	90	84	88	94	103	100	91	91	86	89	93	103	100	90	90	84	88

3.5.2 Evaluation with various defects and process variation density

For different defects and process variation density, most interestingly, set remapping has a very good yield for a small area cost. For example, in the (10, $30mV$) configuration from Table 3(a), 95% set remapping has 100% yield, while 25% redundancy has 94% yield. For one and ten defects per cache from Table 3(a), set remapping achieves 100% yield (the numbers in the table are rounded up from yields as high as 99.9% for 95% set remapping). Yield is higher for line delete because it selects caches that meet the degradation threshold without attempting to cover the faults. Although 12.5% and 25% redundancy with ECC also attains 100% yield, its larger area cost makes it worse than set remapping in terms of YAP. This trend continues for all defect and process variation densities, and organizations.

YAP captures the trade-off between better yield and increased area/decreased performance. The tables show that no redundancy has a reduced YAP as defect density and process variation increase. In some cases, the yield gain is offset by the area cost. Although ECC and redundancy generally improve YAP, it sometimes reduces the YAP metrics with the line delete scheme. This shows the cases where area overhead is greater than the benefit achieved as shown in (1, $30mV$) and (10, $30mV$) of Table 3(a). Another example is (100, $20mV$) in Table 3(b). 25% redundancy with ECC has a better yield than 12.5% redundancy with ECC, but its YAP metrics are lower. Set remapping has the overall highest YAP because yield is improved for a modest performance and area cost. For example, in Table 3(a), 99% line delete in the (100, $30mV$) case has a 66 YAP(M3). In this configuration, set remapping does much better than redundancy and delete, even with ECC.

3.5.3 Evaluation with various organizational parameters

Figure 6 compares three different cache organizations. For the organization in Figure 6(a), one word-line error affects two lines. However, Figure 6(b) has eight unavailable lines with one word-line error. Interestingly, one word-line error in Figure 6(b) affects four times as many lines as in Figure 6(a). One word-line error in Figure 6(c) is four times less than in Figure 6(b). Their bit-line error impacts are the same. Table 3(c) illustrates yield management scheme evaluations for three organizations with the same cache size. Different organizational

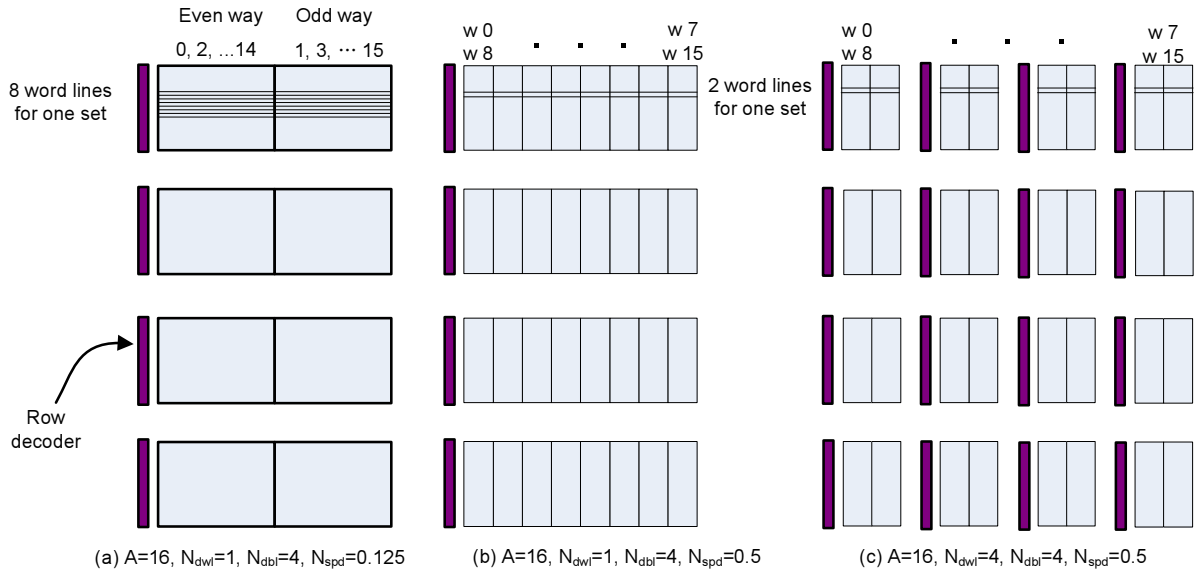


Figure 6: Different organizational parameters.

parameters have different area overheads because of the number of row decoders. For example, the area numbers in the (1, 4, 0.5) configuration from Table 3(c) are different from (4, 4, 0.5) or (1, 4, 0.125). As we can see in the Table 3(c), different organizational parameters have less difference than the fault masking schemes. For each design, the yield differs between 1% to 4% among the configurations. Area difference between each design is not significant. This is because the area occupied by decoders as very small as compared with the area of cell array.

3.5.4 Discussion

DEFCAM simplifies the process to evaluate and select different cache design for a processor architect. With user-defined YAP metrics, computer architects can evaluate many designs for their own purpose. From our evaluations, we make the following observations. First, adding ECC lines to cover hard fault works very well. For different designs including redundant rows and delete schemes, ECC increases yield up to 45%. However, using ECC for hard fault correction makes it unusable to cover soft error. In that case, ECC should be extended

to cover two bit error correction with extra area cost. Next, set remapping achieves the best yield and YAP in most cases, even without ECC. This result shows the benefit of set remapping. Lastly, different organizational parameters lead to different yields and areas. YAP can capture these differences. Nevertheless, in our experiments, the difference is very small; the YAP values differ from 1% to 4% for each metric.

To judge the computational demands of our tools, we measured the speed of DEFCAM by continuously running simulations for 20 hours. With 10 cache designs, 1,000 samples from 10 wafers, and 7 defect-process-organization variation configurations, the design space has 70,000 caches. Using four 3.4GHz Intel Xeon-based Linux boxes, DEFCAM evaluated all caches with no redundancy and redundancy schemes. For evaluating caches with delete and set remapping, DEFCAM could simulate roughly 10% of all caches, i.e., 100 out of 1,000. This sampling results in a maximum error of as low as 0.2% at the 95% confidence interval. This rate is fast enough to evaluate a large design space overnight. The case study demonstrates how our flow and tools can be used to evaluate yield, area and performance of many cache designs. It also highlights the importance that set remapping schemes will play in yield management for future cache designs and technologies.

3.6 DECOUPLED YIELD MODEL FOR CORES AND CACHES

3.6.1 Baseline yield model and parameters

Chip yield is generally dictated by defect density D_0, area A, and clustering factor α. We use a negative binomial yield model from the ITRS report [35], where the yield of the chip die (Y_{Die}) is:

$$Y_{Die} = Y_M \times Y_S \times \left(\frac{1}{1 + AD_0/\alpha} \right)^{\alpha} \tag{3.1}$$

In the above, Y_M is the material intrinsic yield, which we fix to 1 and do not consider in this work. Y_S is the systematic yield, which is generally assumed to be 90% for logic and 95% for

memory [35]. α is a cluster parameter and assumed to be 2 as in the ITRS report. Although technologies with smaller feature sizes are more vulnerable to defects, ITRS targets the same D_0 for upcoming technologies when matured, due to process technology advances.

To compute a realistic yield with equation (3.1) in the remainder of this paper, we derive D_0 from the published yield of the IBM Cell processor chip, which is 20% [64].[3] For accurate calculation, we differentiate the logic portion whose geometric structure is irregular from the memory cell array that has a regular structure in each functional block. While the memory cell array may be more vulnerable to defects and process variability, it is well-protected with robust fault masking techniques, such as redundancy and ECC [35, 60, 48].

We use CACTI version 5.3 [82] to obtain the area of the memory cell array in a memory-oriented function block. From CACTI and die photo analysis, we determined that the memory cell array of the PPE and the SPEs account for about 8% and 14% of the total chip area, repectively.[4] Based on the above analysis, we determine the total memory cell array area is 22% of the chip area ($175mm^2$ in 65nm technology). We can derive D_0 with equation (3.1) using the total non-memory chip area. We calculated D_0 to be $0.0181/mm^2$.

3.6.2 Decoupled yield model

Given multiple functional blocks in a chip and their individual yields (Y_{block}), the chip yield can be computed as [23]:

$$Y_{Die} = \prod_{i=1}^{N} Y_{block_i} \tag{3.2}$$

It is clear that the yield of a vulnerable functional block can be a significant potential threat to the overall yield. Therefore, it becomes imperative to evaluate each functional block's yield separately to prioritize and guide design tuning activities, e.g., implementing isolation points and employing functional block salvaging techniques based on the possibility of the

[3] In Sperling [64] the yield for the Cell processor was vaguely given as 10%–20%. While a lower yield makes an even stronger case for StimulusCache, we conservatively use the highest yield estimate (20%).

[4] CACTI reports that in a 512KB L2 cache (Cell processor's PowerPC element has a 512KB L2 cache) the memory cell array accounts for about 78% of the total cache area. We measure the L2 cache area of PPE from the die photo and use 78% of that to the memory cell array area. The memory cell area of the local memory in SPEs is directly measured using the die photo.

Table 4: Estimated functional block yields of ATOM processor.

Functional blocks	Total area (mm^2)	Logic area (mm^2)	Cell array area (mm^2)	Yield
Front End Cluster (FEC)	2.775	2.425	0.350	95.74%
Integer Execution Cluster (IEC)	0.798	0.798	—	98.57%
Floating Point Cluster (FPC)	1.776	1.776	—	96.86%
Memory Execution Cluster (MEC)	1.897	1.634	0.263	97.10%
Bus Interface Unit (BIU)	2.094	2.094	—	96.31%
Processing	9.340	8.727	0.613	84.85%
L2 Cache	5.318	1.117	4.201	98.01%

salvaging components. To accurately evaluate the yield of individual functional blocks, as suggested in the previous subsection, we propose to define their yield in terms of the logic yield and the memory cell array yield as follows:

$$Y_{block_i} = Y_{logic_i} \times Y_{memory_i} \tag{3.3}$$

Using D_0 derived in Section 3.6.1, we estimate the expected yield for the key functional blocks of the ATOM processor [33] using our decoupled yield analysis approach.[5] Table 4 depicts the area and yield for each functional block. In Table 4, we divide the core into two general functional blocks: processing and L2 cache. The processing block has the five logic-dominant functional sub-blocks (FEC, IEC, FPC, MEC, and BIU). The L2 cache block is a memory-dominant functional block. Although FEC and MEC are logic-dominant functional blocks, they have 32KB and 24KB 8-T (i.e., eight transistors compose one cell) L1 cache. To accurately estimate the functional block yield of FEC and MEC, the 8-T cell array's 30% area overhead over a conventional 6-T cell array is faithfully modeled.

Figure 7 depicts the yield for 8-core and 32-core CMPs using an ATOM-like core [33] as a building block.[6] It separately shows core and L2 cache yield along with the traditional

[5] For various process generations, the initial defect density and the trend of defect density improvement ("yield learning") are very similar [84]. Thus, we can use the derived defect density for 45nm technology without loss of generality.

[6] We assume that the yields of chip I/O blocks and other supporting blocks (e.g., PLL) are 100% for simpler and intuitive analysis. Typically, such blocks employ large geometries, which dramatically decreases the effective defect density. Moreover, we assume that the L2 cache's cell array is salvaged by redundancy and cache block disabling [60]. We employ Monte Carlo simulation [48] to calculate the cell array yield when

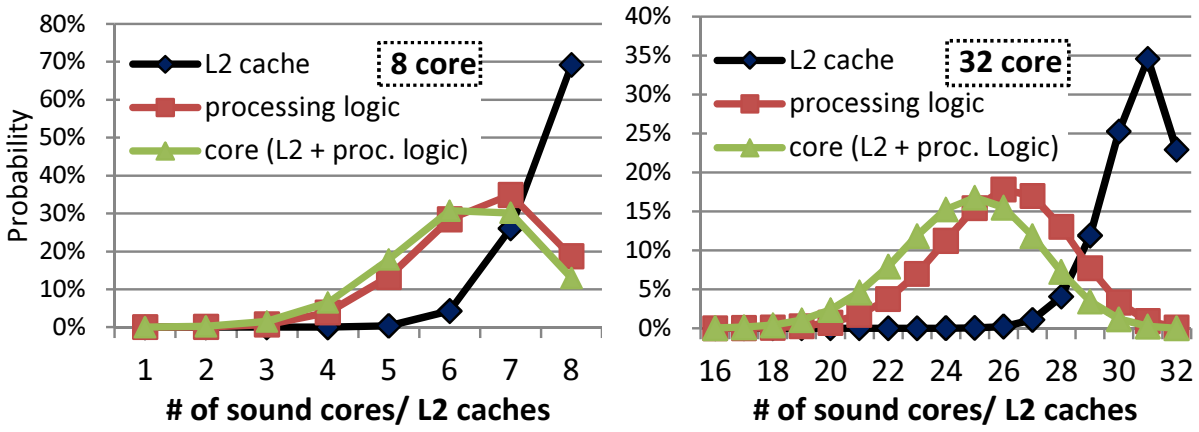


Figure 7: Yield of L2 cache, processing logic, and core (L2 cache + processing logic) for (a) 8-core and (b) 32-core CMPs.

"combined" yield, computed with the decoupled yield model. [7] For the 8-core case in Figure 7(a), less than 13% of the chips have eight sound cores and caches. It is clearly shown that this low yield is caused by the poor yield of the compute cores. In contrast, the cache memory has a much higher yield; in 70% of the produced dies, all eight cache memories are functional. As the core count increases, the discrepancy between the number of sound cores and L2 caches widens. Figure 7(b) shows the 32-core case, where 83% of the chips have at least 30 sound L2 caches while only 5% of dies have 30 sound cores or more.

With core disabling, chip yield can be greatly improved. Figure 8(a) depicts the yield improvement due to core disabling for the 8-core case. We define the criteria for a "good die" based on the core count threshold, N_{th}—i.e., does the chip have at least N_{th} healthy cores? When $N_{th} = 4$, the yield is 91% whereas the raw yield ($N_{th} = 8$) is just 13%. Figure 8(b)(left) shows how many excess caches are available in 1,000 good dies when $N_{th} = 4$ and 4 cores are

such salvaging techniques are used. With 5% row redundancy and disabling of up to 8 lines, the cell array yield is 99.82%.

[7] To get yields for processing cores and on-chip memories, Domer *et al.*'s model [22, 23] and Sarangi *et al.*'s model [75] are used to calculate the impact from physical defects and process variation, respectively. Processing logic and memory model are borrowed from Intel's ATOM processor. Processing logic is assumed to have BIU, MEC, FPC, IEC, and FEC in ATOM processor's core [33] and the area of IO FSB, FUSE, and PLLs is ignored. Processor's floor plan is built up from the manufacturer's die photo, manually calculated core and memory's area is faithful. Inside the processing core, L1 cache and branch predictor's memory area is faithfully modeled. L2 cache's complex logic area, e.g., address decoder, is conservatively assumed to be 10% of whole L2 cache area. 8 spare rows and columns are assumed for L2 cache.

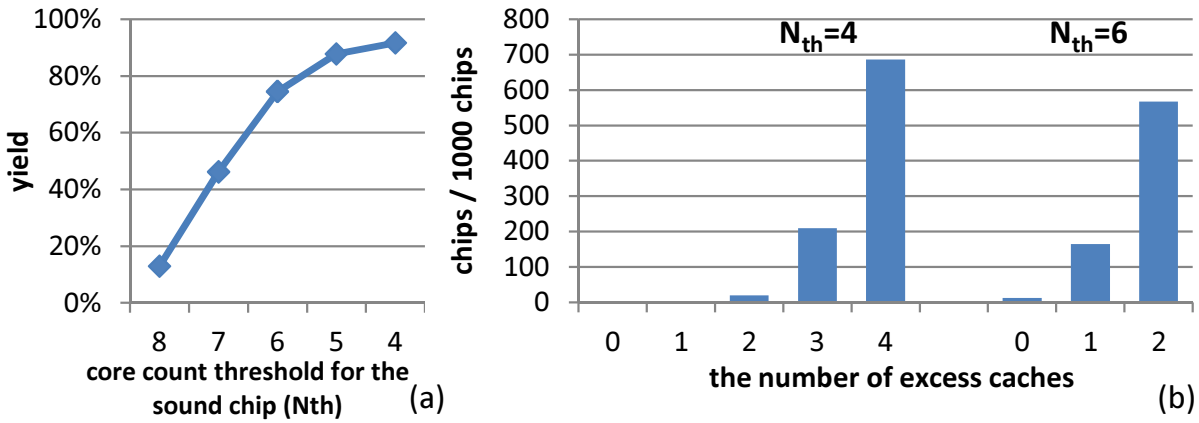


Figure 8: (a) Yield with varying core count thresholds (N_{th}) for the 8-core CMP. (b) The number of chips (out of 1,000 chips) with different numbers of excess caches when $N_{th} = 4$, four cores enabled, and $N_{th} = 6$, six cores enabled.

enabled (i.e., we have two product configurations: 8 cores or 4 cores with excess caches). It is shown that more than 68% of the 4-core chips have four excess caches. Figure 8(b)(right) plots the number of available excess caches when $N_{th} = 6$ and 6 cores are enabled. Well over half of all 6-core chips have two excess caches. These results demonstrate that there will be plenty of excess caches from the loss of faulty cores in future CMPs. Once tapped, these unemployed, virtually free cache resources can be used to improve the performance of CMP systems.

3.7 SUMMARY

Traditional cache optimizations focused on the performance, area, and power aspect of the resulting design. Guaranteeing fault tolerance and enhancing chip yield have been largely a separate effort made by low-level circuit quality engineers, layout designers, and process engineers. As chips built with a future deep sub-micron technology are more susceptible to manufacturing defects, process variations, and aging phenomena, it becomes imperative to consider reliability and yield together at an early design time by the processor architect/cache designer.

This chapter presents DEFCAM, a new cache design flow that integrates cache defect, yield, and performance models, and lays a solid foundation for evaluating a cache memory designed on nanometer-scale technology in terms of its yield, area, and performance. A metric called YAP (yield-area-performance) is introduced, which enables a designer to quickly evaluate different cache designs with a single number. Using DEFCAM and the YAP metric, a cache designer can directly compare cache designs that have different architectural, organizational, and defect-related parameters at an early design stage.

4.0 ADDRESSING YIELD FOR L1 CACHE

4.1 FAULT MASKING STRATEGIES: DELETE SCHEMES

DEFCAM can model many different fault schemes for cache memory as described in Section 3.3. In this section, we use DEFCAM to explore some of these schemes, as well as new ones. We start with "delete schemes" that disable portions of the cache. These schemes are already present in some modern processors [74, 14] and are based on the observation that programs run correctly as long as the processor retrieves (from a defective cache) correct data regardless of cache access latency. However, any performance degradation due to masking the defects is a critical issue for processor usability and yield analysis/management. There are general strategies that can be used to trade performance for guaranteed correct operation, including line, set, and way delete.

4.1.1 Line delete

When a particular cache line is faulty, it can be marked and excluded from normal cache line allocation and use. A programmable *fault map* can be provided to record the markings. As an implementation of the fault map, an "availability bit" may be attached to each cache tag and treated as second valid bit [57]. Once the system is turned on, a memory BIST engine performs testing, and sets the availability bits. Any cache line with the availability bit turned off is faulty and not used. A similar strategy has been employed in IBM's Power4 and Power5 processors, which can delete up to two cache lines in each L3 cache slice [14]. For this scheme to work, cache line allocation (e.g., LRU or FIFO) has to be aware of the fault locations in a target set. Several circuit techniques have been studied for such LRU

logic [42, 56].

4.1.2 Set delete

When a set becomes faulty, it can be marked as deleted. In certain cases, set deletion can be done by deleting all lines in the set if a line delete scheme is employed. On the other hand, since the nature of faults may not allow per-line or per-set fault marking schemes utilizing the tag memory (e.g., a specific entry in the tag memory is not accessible), more robust fault map techniques may be needed. One such technique is to employ a second decoder leading to an array of fault map bits.

4.1.3 Way delete

A cache way can be shut down if it becomes unavailable due to defects. Conceptually, an N-bit fault map can tell which ways are unavailable in an N-way set-associative cache. Depending on the nature of defects, a cache way may be shut down by simply turning off a specific per-line availability bit in all cache sets. As in the case of set delete, a more robust fault map scheme than a per-line available bit scheme may be needed. For instance, one may introduce N faulty way bits and on each cache access, treat those bits as the third valid bit (just like the per-line availability bit), common to all sets. There are also microarchitectural techniques to shut down cache ways to save power consumption [3], which can also be used to delete defective cache ways.

4.1.4 Evaluation methodology: greedy algorithm

We used DEFCAM to assess the impact of line and set delete schemes. We do not include the impact of disabling a way because it does not have significant impact except when the whole cache is turned off [47]. For more informative assessment, DEFCAM needs to evaluate the best- and worst-case scenarios for line and set deletion. For example, the deletion of a frequently used cache line may lead to different performance than the deletion of a rarely used line. Thus, we develop an algorithm that finds the worst/best miss counts (miss rate)

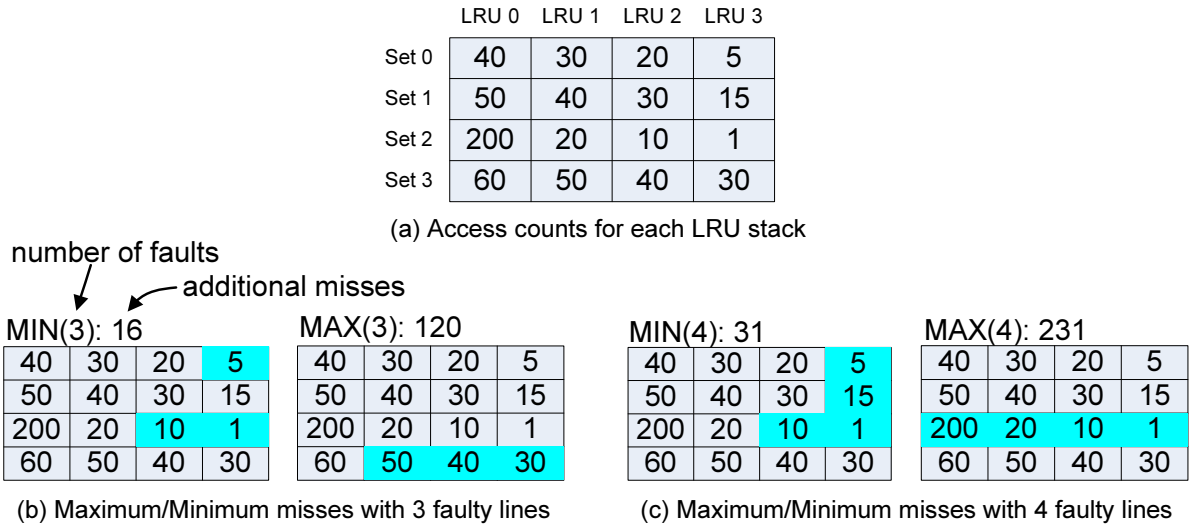


Figure 9: An example access count profile and fault maps leading to most or fewest misses.

when the most/least frequently used lines or sets are deleted. Figure 9 illustrates how the algorithm works for a four-way set-associative cache with four sets. The figures show the number of accesses per cache line for an example workload. Figure 9(a) shows the access counts for each set and LRU stack without any faults. LRU 0 means the most recently used line in the LRU stack. For example, the number for LRU 0/Set 1 is the total hit counts from LRU 0 stack in set 1 (50 from the figure).

In Figure 9(b), MIN(n) shows the minimum additional misses that result when there are n faulty lines. The minimum is computed based on the best case (i.e., the least heavily used lines are faulty). For example, "MIN(3):16" means there are only 16 additional misses when three lines are faulty, given the access counts from Figure 9(a). The highlighted lines are the faulty ones. Similarly, MAX(n) shows the maximum additional misses with n faults.

To find MIN(n), the algorithm simply accumulates the small n numbers from the given access counts. For example, to find MIN(4) , the next smallest line access count (15) is added to MIN(3). On the other hand, determining MAX(n) is more complex. For MAX(n), the algorithm sums the access counts for each set and picks the set with the highest summation. For each set, the summation starts from the LRU block to the MRU block. For example, MAX(3) is the summation of LRU 3, 2 and 1. As shown in Figure 9(b), set 3 has the

Table 5: Key machine parameters.

Benchmarks	Parameters
MiBench	**"ARM based Embedded Processor"**
	Single in-order pipeline 8kB 16-way I/D caches – 32B line, 1-cycle latency 50-cycle latency main memory via a 64-bit bus 2k-entry bi-mod branch predictor
SPEC2000	**"High-Performance Superscalar Processor"**
	8-issue out-of-order processor with 128 ROBs 32kB 4-way I/D caches – 128B line, 3-cycle latency 2MB 8-way L2 cache, 256B line size, 18-cycle latency 240-cycle latency main memory via a 128-bit bus 4k-entry combined branch predictor

maximum number of summation from LRU 3 to LRU 1. However, MAX(4), as illustrated in Figure 9(c), comes from set 2 because LRU 0 block of set 2 has far greater number than that of other sets.

Currently, DEFCAM generates cache access profile data using SimpleScalar with an additional cache profiler module. The impact of the delete schemes for each benchmark is produced with the greedy algorithm in Section 4.1.4. Table 5 summarizes the experimental configurations. We selected these configurations to represent two design points—one for embedded systems and the other for high-performance systems. The embedded processor is modeled after an ARM-based design and the high-performance processor after an aggressive out-of-order superscalar design. We used different benchmark suites for each processor to represent the kinds of applications that would run on these designs. The embedded design uses MiBench [28] and the high-performance design uses SPEC2000. Simulation benchmark configuration is summarized in Table 6.

Figure 10 illustrates the relationship between miss rate (y-axis) and capacity loss (x-axis) due to two delete schemes: line delete and set delete. Left four figures are for line delete and, similarly, right four are for set delete. It shows only four benchmarks for brevity; two are from MiBench [28] and two are from SPEC2000. These benchmarks have two extreme cache access trends. One case has uniformly distributed accesses to all lines/sets and the

Table 6: Benchmark simulation configuration.

Benchmarks	Inputs	Fastforward	Warmup	Simulation period
MiBench	Small input sets	No	No	Whole execution
SPEC2000	All inputs, Averaged	5B instructions	500M instructions	1B instructions

other case has a bias toward accessing only certain lines/sets. The balanced cases are *mgrid* and *cjpeg*. The biased cases are *vortex* and *dijkstra*.

Each graph has three curves showing the maximum, average, and minimum performance impact. As shown, *vortex* in Figure 10(a) and *dijkstra* in Figure 10(b) have a large gap between the maximum and minimum curves. These "big eyes" suggest that cache set usage is heavily unbalanced and clustered in these programs and that only a few lines are actively used in the sets. Therefore, it becomes more difficult to predict performance accurately given the number of faulty cache lines or sets for these benchmarks. On the other hand, *mgrid* in Figure 10(a) and *cjpeg* in Figure 10(b) have a narrower gap between the curves, especially in the set graph. *cjpeg* again has the most balanced usage of cache lines within each set, and the maximum impact curve for line deletion becomes in essence a 16-segment (16-way cache) piecewise linear curve.

Now, we turn our attention to the criticality of the lines (sets) deleted. Table 7 summarizes the maximum, average, and minimum miss rate with 12.5% capacity loss as an example for the four programs in Figure 10. Interestingly, the maximum miss rate from the line losses and the set losses are the same for all four programs regardless of the program's characteristics. We can interpret that those faulty lines compose faulty sets which have the same amount of capacity loss. Consequently, even with 12.5% disabled lines, the miss rate can be more than 50% for programs with "big eye" from Figure 10. "Small eye" programs also have significant miss rates (i.e. more than 29%) when critical lines become unusable.

For average and minimum miss rates, faulty sets generate a moderate miss rate, while faulty lines have a small miss rate. On the average case, faulty sets' miss rate is almost corresponding to the amount of the capacity loss. Minimum miss rate row in Table 7 illustrates that set faults produce a substantial miss rate even if they make the least number

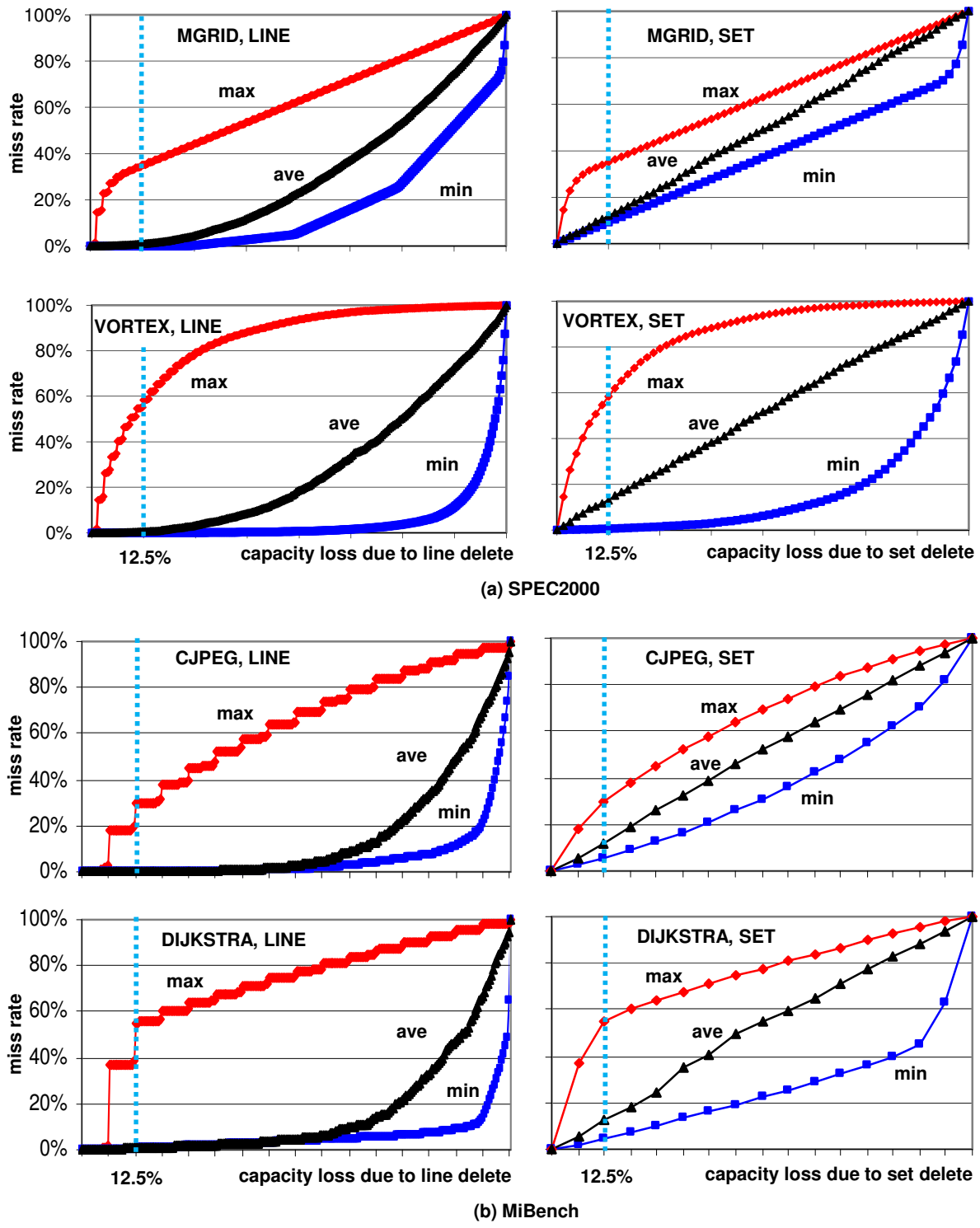


Figure 10: Max./avg./min. impact of deleting lines (left) and sets (right) on miss rate. for selected programs.

Table 7: Miss rate with 12.5% capacity loss for selected programs.

	MGRID	VORTEX	CJPEG	DIJKSTRA
MAX:line	0.3506	0.5842	0.2949	0.5535
MAX:set	0.3506	0.5842	0.2949	0.5535
AVE:line	0.0090	0.0064	0.0006	0.0065
AVE:set	0.1216	0.1296	0.1140	0.1267
MIN:line	0.0000	0.0006	0.0004	0.0044
MIN:set	0.0927	0.0053	0.0528	0.0471

of misses with the given capacity loss. However, the minimum impact of line faults is very limited. For example, *mgrid* shows 9.27% miss rates for minumum set faults while miss rates becomes negligible for minumum line faults. In fact, *mgrid* has significant miss rates from only one faulty set (see Figure 10), which leads to sizable performance penalty. From these observations, we conclude that set fault is the most important to mitigate performance penalty, whereas line fault is much less significant. Line faults only become meaningful when they compose set faults.

4.1.4.1 Experimental results Figure 11 shows the misses from one set fault using a delete scheme for all programs in MiBench and SPEC2000. The graph shows the additional misses caused by deletion per 1,000 instructions. As we can see in Figure 11, many programs have a large number of additional misses only from one faulty set. For example, *mcf* in SPEC2000 generates 97 additional misses per 1k instructions with only one faulty set. Similarly, *tiff2rgba* and *rijndael* in MiBench add more than 70 misses per 1k instructions with one faulty set. Although delete schemes are used in many processors, they cannot guarantee performance when an entire set is deleted.

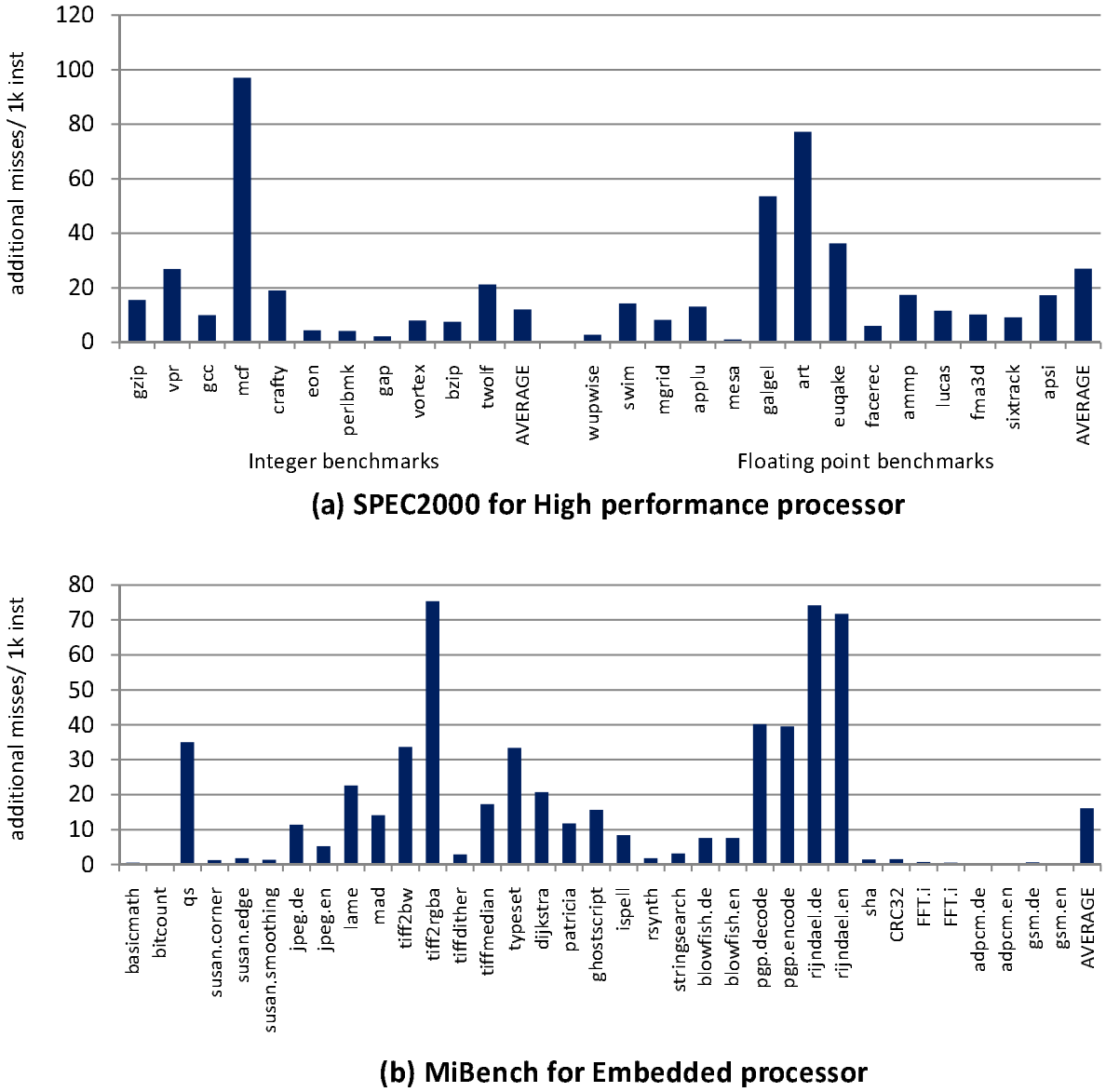


Figure 11: Additional misses per 1,000 instructions with one faulty set defect.

4.2 FAULT MASKING STRATEGIES: SET REMAPPING SCHEMES

From the above study, we have two findings. First, the additional misses can be limited unless faulty lines compose fault sets. For all MiBench programs, the average additional misses from 50% faulty lines is limited to 3% from our study. Second, however, the loss of a small portion of the address space from set faults makes a significant performance penalty if it can not be cached.

Based on this observation, we propose *set remapping*. When cache sets (and the cache lines in them) are damaged, accesses to the faulty sets can be directed to other sound sets. Figure 12 shows how a conventional row decoder can be changed to accommodate this strategy. The proposed set remapping scheme calls for a change in the decoder driver (typically a series of inverters) as shown in Figure 12(a) and (c). In addition, a set of programmable "remap match" registers are needed to record the faulty cache sets to remap (Figure 12(b)). When there is a cache access to a "remapped" set, one of the remap registers has a match, and consequently, drives the wordline to the sound (replacement) target set.

The performance impact of set remapping depends on the target set chosen for a faulty set. Well chosen target sets can reduce the cache miss count dramatically. How can the best target set be picked for a faulty cache set? A heavily used set is not expected to be appropriate. However, if most accesses on that target set use only a few lines among all lines in the set, even a heavily used set can be an appropriate target set. Furthermore, it is possible that the target set's usage is limited to the specific program phase when the faulty set is rarely accessed. In these cases, using access count numbers for each set to find a target set does not guarantee its optimality. Therefore, we investigate several heuristic approaches to finding the best target set. We consider three set remapping schemes: static, profile-based, and dynamic remapping.

4.2.1 Static remapping

This scheme selects a sound target set for a faulty set at design time (i.e., it is hard-wired at manufacturing time). The target sets that can be used for a faulty set are pre-defined at

design time. An access to faulty set is remapped to a specific target set. Thus, a target set handles accesses which would have been to two different sets (i.e., the remapped faulty set and the target set). To select a target set for a faulty one, we randomly pick a replacement from the candidate targets in a round-robin fashion, starting from the first target set. This selection avoids the need for information about program behavior. If there are more target sets than faulty sets, one target set index shares at most two set indices.

To implement this scheme, we need two hardware changes. First, an additional "fault bit" must be added in the tag memory to distinguish the original set index and the remapped set index. If the design only supports the case where the number of faulty sets is smaller than the number of target sets, only one "fault bit" is required. When the processor accesses a faulty set that is remapped to a target set, the fault bit is "1." Otherwise, an access for a non-remapped set has a "0" in the fault bit. If there are more possible faulty sets than target sets, more faulty bits are needed, which is not assumed in this study. The second hardware change is to include a remap-enabled decoder (shown in Figure 12). The access index number is compared to the remap register value which has the fault set index number. If it matches, the appropriate target set is accessed instead of the faulty set. The mapping between faulty and sound target sets is done at manufacturing time. The remap decoder is permanently programmed at this time.

The addition of the remap capability is not expected to have a performance penalty in access latency. The necessary additional logic has two components: comparators in the remap unit and NOR gates on wordlines as shown in Figure 12(b) and (c). Each comparator has a data storage and a XOR logic. The data storage has a fixed value that is set with laser fusing during manufacture. XOR logic has four transistors among which only one transistor causes switching delay. Therefore, the comparison latency of the remap unit in Figure 12(b) consists of NMOS transistor delay and AND gate dalay. A conventional address decoders' critical path has multiple NAND and NOR gates to reduce the latency of the fan-in delay of large fan-in gates [88]. Therefore, address matching in the comparators is faster than normal address decoding and the comparison latency can be effectively hidden. NOR gates are used to select an actual target set on a match and they replace inverters on the potential target sets. Again, with proper circuit optimization (e.g., cell sizing), the use of NOR gates rather

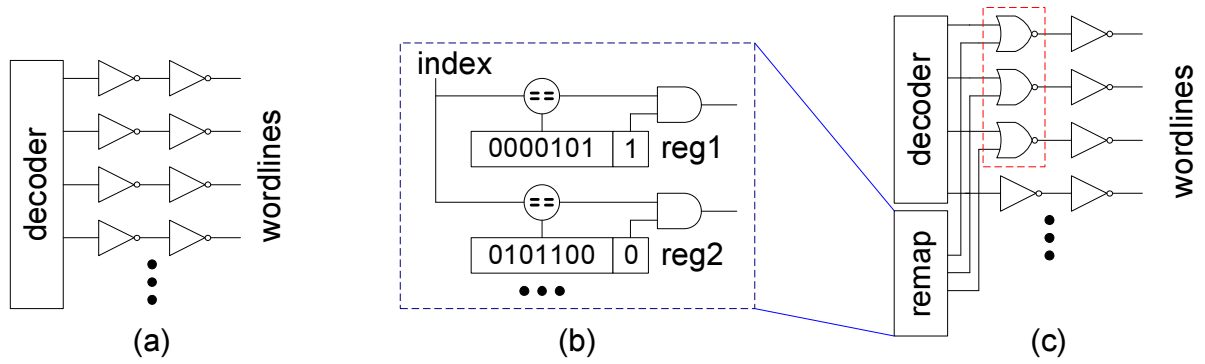


Figure 12: (a) Conventional decoder. (b) Remap unit. (c) Remap-enabled decoder.

than inverters leads to no performance penalty.

4.2.2 Profile-based remapping

Although static remapping covers the whole address space, it may not select the best target set because it does not use a profile about the expected workload to pick target sets. Some applications may have many memory accesses in a single set while other applications have evenly distributed access patterns. Utilizing profile data for each application gives a good clue to decide the most appropriate target for each faulty set. The profile includes access count and miss count. Two phases compose this scheme: the profile collecting phase and the actual execution phase. First, a processor is simulated assuming all sets are good to collect access patterns before the actual execution of the application. Counters for each cache line are used to get access patterns. Miss counts or access counts can be used for the whole application execution. This phase collects profile data for all possible faulty sets. Next, the best target set can be decided for the specific faulty set. The best target sets are the ones which produce the least additional misses counts from remapping. Once target sets are established for the application, this information can be used at run time directly.

We suggest three strategies to choose the best target sets. First, the best target set is chosen with the minimum hit count. When a faulty set is remapped to a target set, maximum additional miss count is limited by the original hit count in target set. Therefore, choosing

the set with the smallest hit count will minimize additional misses. Second, the minimal access count set is picked for the target set. Assuming the miss count is the proportion to the access count, the set with the least access count can be a good candidate target. Lastly, the last half of the LRU stack access count could be used as the candidate. When one physical set shares two sets (the target set and the faulty set), higher LRU stack accesses for the two sets may not contribute additional misses. For example, in four way set associative cache, two higher LRU stack accesses (i.e., four LRU stack accesses in total) rather than two lower LRU stack accesses for each of the two sets will have more chance to be safe from being evicted. Consequently, lower LRU stack accesses will have more chance to sum up more additional misses. Based on the assumption that LRU stack access counts are balanced between the target set and the faulty set, the lower half of LRU stack access is counted.

4.2.3 Dynamic remapping

Although profile-based remapping uses actual access information to determine target sets, once the targets are chosen, it can not adapt the targets based on the actual workload. It also requires the workload be executed once for profiling. Therefore, a question can be raised: What if applications have significantly different access patterns for different execution phases? Benefits will be limited even if we use access patterns to determine appropriate target sets.

We suggest to gather online profile data during program execution. Applications have many different phases during execution and each phase may have significantly different memory access patterns. Therefore, up-to-date information from a program's current phase can lead to more accurate set selection. Utilizing current data requires dynamically programmable address decoder in the cache. *A time window* is defined as the period of execution in which data is accumulated. Target sets are reassigned from the current profile data in the beginning of the new *time window*. After determining target sets, all information is reset to '0' to be ready for accumulating the next phase's information. Whenever a new time window begins, there are additional cache misses for two indexes, one for the old target set and the other for the new one. Therefore, *dynamic remapping* may have worse performance

than static or profile-based remapping. In this case, instead of dynamic remapping, profile-based or static remapping is preferred. If the application has a similar access pattern for its whole execution, the target sets determined by dynamic remapping will be similar to static or profile-based remapping.

4.2.4 Experimental results

Using DEFCAM, we examined the three remapping schemes for the same processor models and benchmarks as the delete scheme study. Figure 13 compares the schemes for all MiBench benchmarks on an ARM-based embedded processor. Figure 14 does the same comparison for all SPEC2000 benchmarks on the superscalar processor. Figure 13(a), (c), 14(a) and (c) show the misses for static remapping normalized to the delete scheme. "Fault free" is the number of misses without any fault. Figure 13(b), (d), 14(b) and (d) introduce the additional miss reductions for profile-based and dynamic remapping per 1,000 instructions. "Full" and "4" are the number of candidate target sets. In other words, full can choose any set as a target set whereas 4 has only 4 candidate target sets. Oracle chooses the best target set for each time window and shows the minimum misses for dynamic remapping. Because dynamic remapping must flush data for a dirty line that is remapped, a smaller window size is not always good. After sample simulations, we determined 1M cycle window size is the best choice for our benchmarks. We simulated many possible combinations between faulty sets and target sets and averaged them to get these numbers. As shown in Figure 13(a), (c), 14(a) and (c), static remapping can reduce most of the additional misses arising from the set delete scheme. Although static remapping for *stringsearch* incurs significant additional misses relative to the set delete scheme in Figure 13(c), *stringsearch* has a very small misses for the delete scheme in Figure 11(b). Therefore, the role of dynamic or profile-based remapping for *stringsearch* is very limited. In a similar fashion, static remapping reduces most of the additional misses of the delete scheme for SPEC2000 benchmarks as shown in Figure 14(a) and (c). Although *wupwise* and *sixtrack* in Figure 14(c) have a noticeable gap which cannot be eliminated by static remapping, Figure 14(d) shows that dynamic or profile-based remapping scheme reduces small additional misses (i.e., less than 0.11 per 1k instructions). Therefore, the

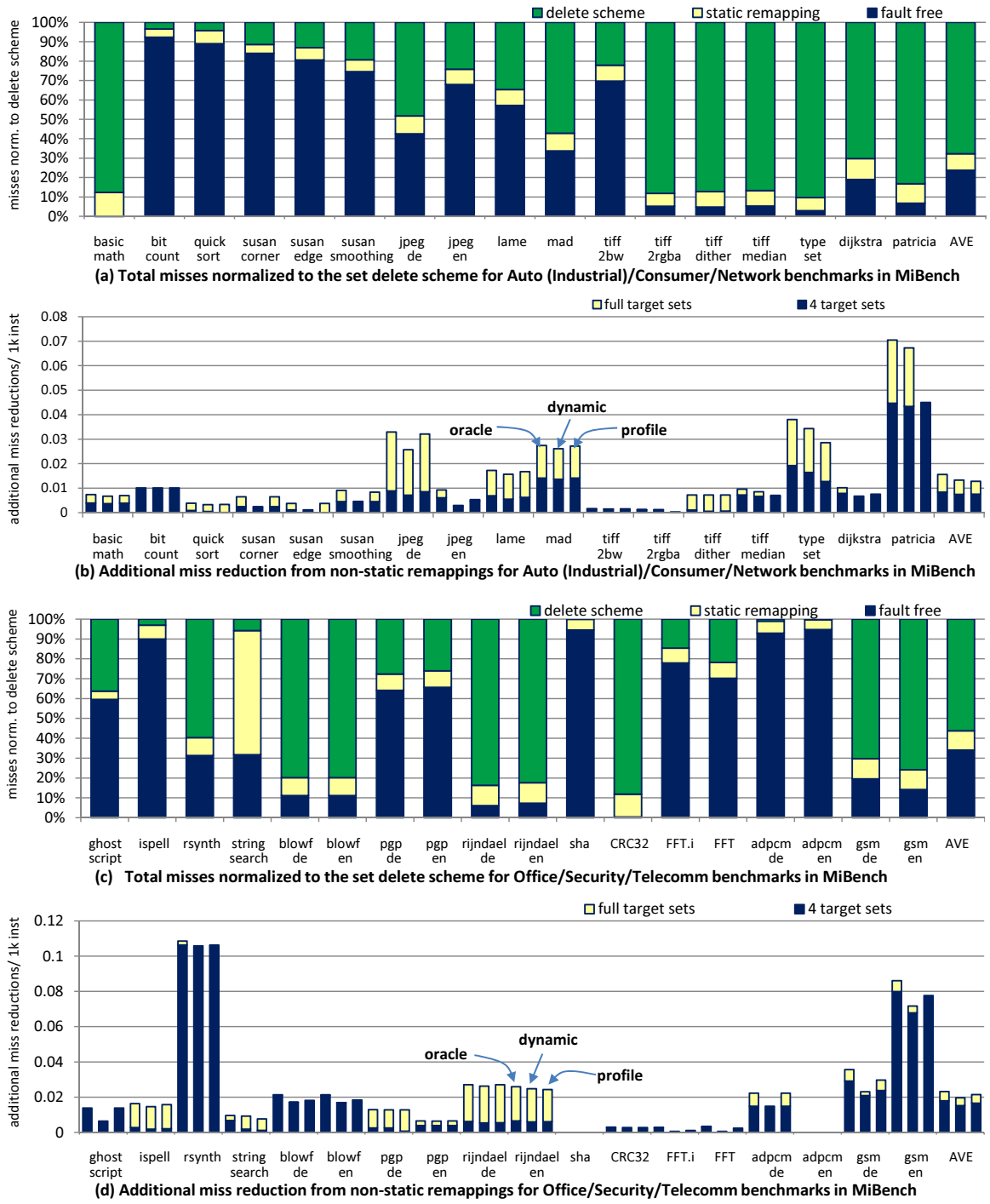


Figure 13: Total misses normalized to the set delete scheme and additional miss reductions from three "non-static" remapping for MiBench; (a) and (b) are for Auto (Industrial)/Consumer/Network benchmarks, (c) and (d) are Office/Security/Telecomm benchmarks; oracle in (b) and (d) shows the theoretical limit of dynamic remapping (i.e., it knows the best target sets whenever new target sets are assigned). Dynamic remapping predicts the best target and assigns it to the next execution window (e.g., 100M instruction interval) from the information of the current execution window. Profile collects 'profile' information in the "phase-collecting phase" and assigns best target sets for whole execution time (i.e., it becomes static remapping after assigning target sets).

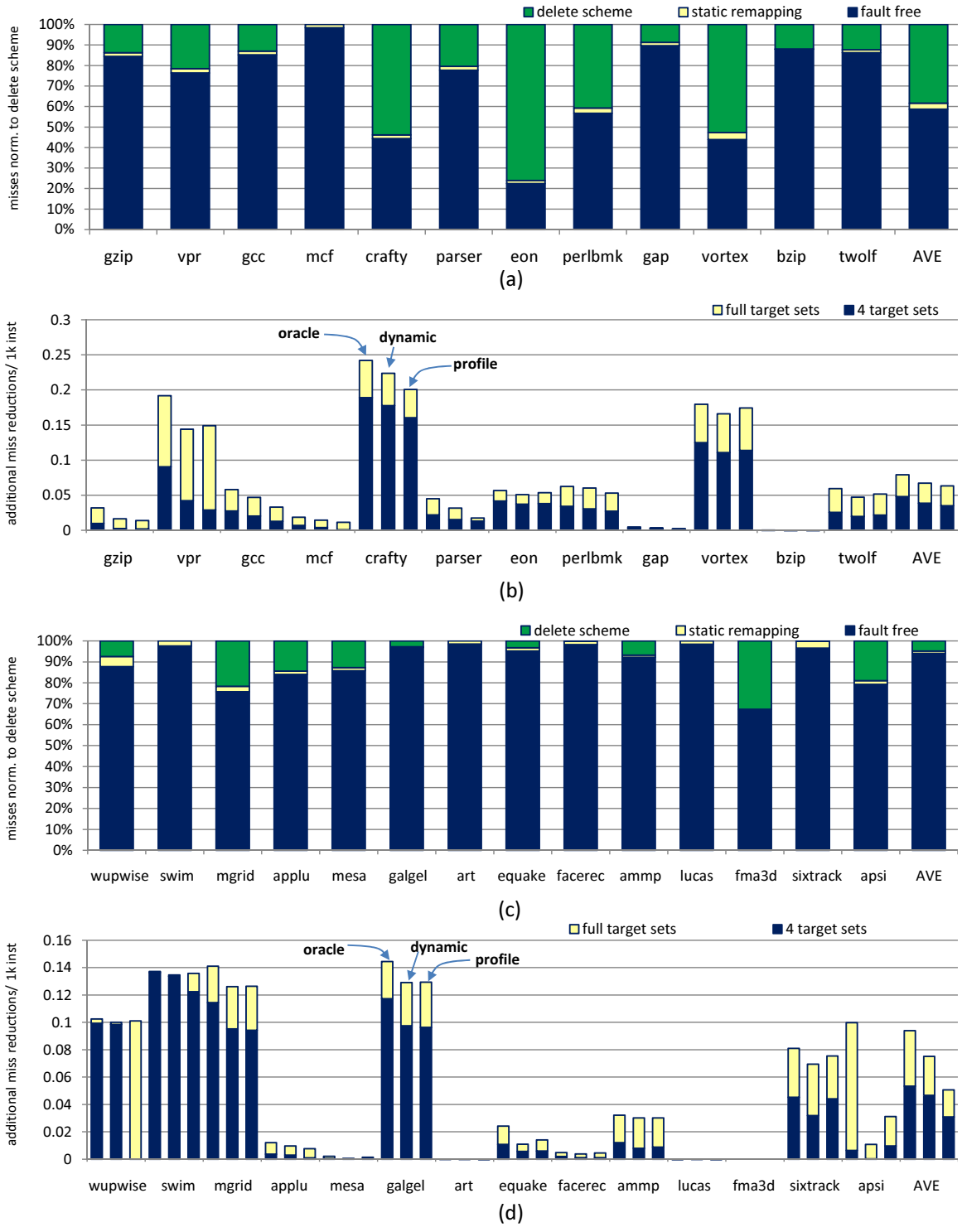


Figure 14: (a) and (b) are for SPEC2000 INT; (c) and (d) are for SPEC2000 FP. the meaning of oracle, dynamic and profile follows the notation in Figure 13.

benefit of using dynamic or profile-based remapping for these applications is not significant. From these experimental results, we conclude that static remapping is the best choice. It reduces the misses over the set delete scheme but does not incur the overhead of profile-based or dynamic remapping.

4.3 SUMMARY

This chapter illustrates how DEFCAM can be used in practice. A case study is performed to compare a number of cache yield management strategies, including: redundant row, ECC, and line delete. When the redundant row and ECC schemes are used in isolation, they fall short of other degradation-based schemes quickly as the effect of defect and process variations is increased. Degradation-based schemes such as line delete were shown to offer much higher yield with limited area overheads.

With DEFCAM, this chapter evaluates the performance impact from three architecturally visible fault classes (line, set, and way) and discovered that masking set faults is crucial to reducing cache misses, and hence, minimizing performance impact. To tackle set faults, set remapping is proposed, which redirects memory accesses to faulty sets to available sound cache sets. This chapter also shows that set remapping can be done statically or dynamically, using off-line or on-line profile information. Among these remapping policies, the static remapping policy is the simplest in terms of design complexity and it achieves much of the potential of an oracle approach.

5.0 ADDRESSING YIELD FOR ON-CHIP DIRECTORY

In this section, we describe the on-line fault detection and correction algorithms for the cache coherence directory. We will also discuss its hardware controller design issues.

5.1 HARDWARE MODEL

We assume a tile-based CMP architecture that employs a MESI-based coherence protocol as our baseline design. Our coherence protocol is similar to the one used in the SGI Origin2000 [43], whose directory structure and states are depicted in Figure 2.3. The SGI Origin2000 was a ground-breaking multiprocessor system that used a directory-based cache coherence protocol. The details of the protocol have been well documented [58]. We assume that the state field for the L2 cache's directory has three bits, two for three basic states (i.e., E, S, I states) and one for corresponding three busy states. The sharer field uses the full-map encoding method and records which cores share a block using an N-bit vector for an N-core CMP configuration.

In this chapter, our main focus is to protect the sharer field in the coherence directory because it is much larger than the state field in a large-scale CMP. The sharer field size also grows with increasing core count and cache capacity. We assume that the state field is protected with other design methods such as redundancy, ECC, or special and hardened memory cells [85]. Thus, we do not consider fault occurrences in the state field.

We use a bit-level fault model where bits in the directory memory are independently struck by a hard error, a transient error, or a soft error. Hard errors are caused by defects and severe process variations. Random process variations are difficult to control in advanced

process technologies, especially in sub-45nm regimes, and can become a dominant source for many memory cell failures. Transient errors occur to weak memory cells when the chip's operating conditions change, such as voltage, frequency, temperature, or lifetime. We classify a hard or transient error as either a "stuck-at-1" or "stuck-at-0" error. Once a stuck-at error occurs to a memory cell, its content is fixed to 1 or 0 thereafter. The severity of error occurrences is measured with HER, which is the ratio of the failed memory cell count to the total memory cell count. Equivalently, HER is the probability of having an error in a memory cell. A soft error may flip a memory cell's content from 0 to 1 or 1 to 0, but does not affect the functionality of the cell. We use failures-in-time (FIT, 10^6 hours) as the rate of soft error occurrences to a given memory block.

5.2 HARD ERROR DETECTION AND CORRECTION STRATEGIES

We make two key observations that are useful for tolerating faults in the coherence directory. First, when a cache block is in the exclusive state, the block's directory entry sharer field normally has only a single bit set (to value "1") and all other bits reset (to value "0"). Effectively, the encoding in the sharer field in this case is a "one-hot" coding, which is very sparse. Hence, we can extract exploitable redundancy in the sharer field if we employ a denser coding method than one-hot. Second, when a block is in the shared state, updating the sharer field involves one of two possible atomic actions: (a) One bit is set (from 0 to 1) when adding a new sharer, or (b) all bits are reset to 0 when invalidating the corresponding cache block. A latent fault in the sharer field (e.g., failure to set a bit in action (a) because the bit is stuck-at-0) may turn into a program error if the directory controller is unaware of the fault and fails to send an invalidation message to the corresponding core in action (b). Fortunately, there is an opportunity to avoid the error situation, if the directory controller is capable of detecting a hardware fault before taking the action (b), by speculatively invalidating potential sharers. Such a conservative, speculative invalidation strategy may degrade program performance if speculation is wrong–i.e., there was no error caused by the hardware fault.

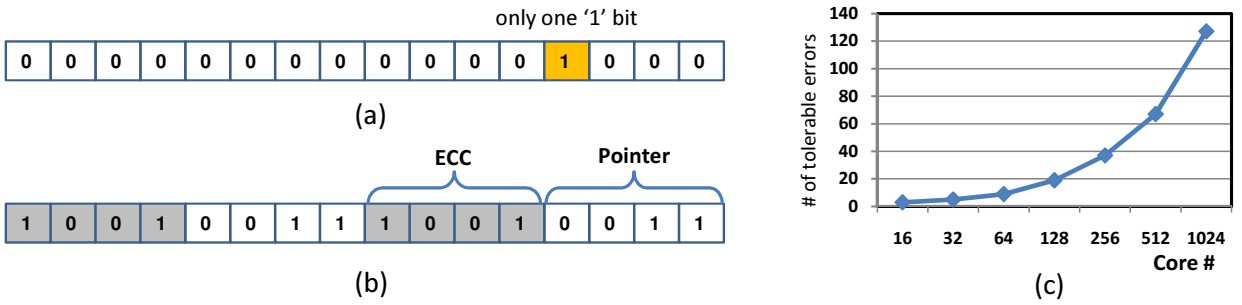


Figure 15: (a) Original sharer field encoding example (16 cores). (b) The proposed ECC-pointer pair encoding for the same example. (c) The number of tolerable errors in one directory entry using the proposed ECC-pointer pair encoding method grows with the number of cores.

5.2.1 Protecting exclusive directory entries

Based on the first observation above, let's consider how directory entries in the exclusive state can be protected. The key idea is to binary-encode the core number (i.e., pointer) instead of using the one-hot coding. We "squeeze" bits that can be used to embed ECC. Figure 15(a) and (b) depict how the sharer field is encoded using the conventional one-hot coding method and our ECC-pointer method, respectively. With the embedded ECC, when there is a bit error in the sharer field, the correct core number can be restored. In the figure, we showed a 16-core CMP example. Hence, a pointer requires four bits and its ECC is four bits. In general, pointer encoding requires $\lceil \log_2 N \rceil$ bits given N cores. There are $(N - \lceil \log_2 N \rceil)$ remaining bits in the sharer field. Provided that we use SECDED to protect an encoded pointer, we need $(\lceil \log_2 \lceil \log_2 N \rceil \rceil + 2)$ bits. Therefore, an ECC-pointer pair requires $(\lceil \log_2 N \rceil + \lceil \log_2 \lceil \log_2 N \rceil \rceil + 2)$ bits.

To further increase error correction strength, multiple ECC-pointer pairs can be embedded using our method, as in Figure 15(b). We have space to record two ECC-pointer pairs in this example. The proposed ECC-pointer encoding is capable of correcting three errors and detecting four errors when the errors are evenly distributed to the upper and the lower halves of the directory entry. When more than three errors concentrate in one half, either upper or lower, straightforward error correction is not possible because the SECDED code cannot differentiate the erroneous half from the error-free half. In this case, one may inspect

each bit (e.g., to see if it's stuck-at-1 or stuck-at-0) to perform errata-based error correction. We do not pursue such a strategy because the probability of this case is extremely low. One may also choose to disable heavily faulty directory entries (and cache blocks) by using a degradable cache mechanism [61, 47].

The proposed encoding strategy the desirable property of correcting more bit errors as the number of cores increases. Figure 15(c) depicts how the number of tolerable errors grows with the number of cores. For example, when there are 256 cores, 19 ECC-pointer pairs can be embedded, which provides more than 37-bit error recovery within a directory entry. Note also that the proposed strategy works equally well to combat hard errors, intermittent errors, and soft errors. Because the proposed encoding strategy is based on ECC, we do not need separate procedures to detect and correct errors for directory entries in the exclusive state. For errors that occur to directory entries in the shared state, however, we need separate error detection and correction steps.

5.2.2 Protecting shared directory entries

Our second observation forms the basis for protecting directory entries in the shared state: An error may propagate when the directory controller is unaware of the error and fails to deliver an invalidation message. Hence, one would be able to guarantee correct program operation *if the directory controller is capable of detecting errors and sends out invalidation messages speculatively and conservatively on detecting errors.* This implies that the error detection and correction for shared directory entries need to be done at invalidation time. Nevertheless, to offer soft error immunity, PERFECTORY detects faults and updates directory entries whenever there is a read or update event to a shared directory entry. Soft error immunity is discussed in Section 5.3.

We propose a proactive on-line error detection algorithm for sharer bits using a simple bit inspection method. To detect hardware errors in a directory entry, PERFECTORY performs on-the-fly test of the entry by applying bit patterns to the entry and reading from it. For example, a stuck-at-1 error is detected when "0" was written and "1" was read. Similarly, a stuck-at-0 error is detected when "1" was written and "0" was read. Our detection algorithm

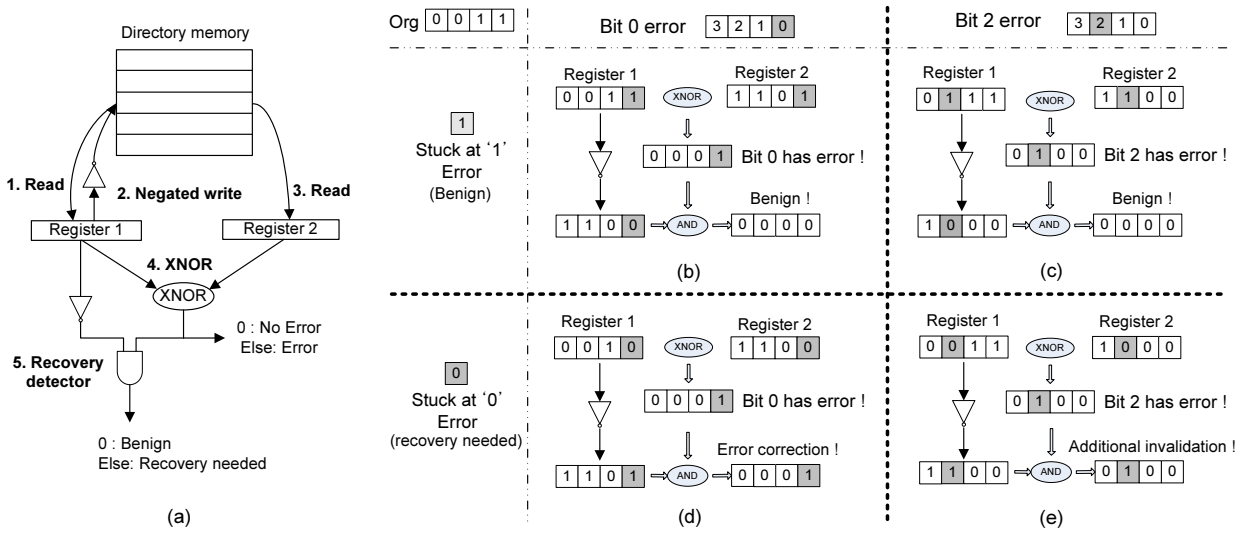


Figure 16: Read-time detection algorithm. (a) Hardware architecture. (b) Example of a benign stuck-at-1 error. (c) Example of a stuck-at-1 error causing a speculative invalidation message and potential performance degradation. (d) Example of stuck-at-0 error that lends itself to correction. (e) Example of benign stuck-at-0 error causing a speculative invalidation.

can detect any number of hard and intermittent errors and is exercised whenever a directory entry is read ("read-time detection"). Figure 16 shows how the read-time detection algorithm works.

Our on-line read-time detection algorithm works in four steps. First, the directory controller reads the sharer field of a directory entry whenever the entry is referenced, e.g., to invalidate a cache block. The read value is saved in a temporary register–Register 1 in Figure 16(a). Second, the value in Register 1 is "negated" (i.e., bits are flipped) and is written back to the sharer field. Note that the original value in Register 1 is intact. Third, the directory controller reads out the same entry once again and stores it to Register 2. Lastly, the controller finds hardware errors by comparing the contents of Register 1 and Register 2 using a bit-by-bit comparator ("XNOR" logic). A specific bit position with "1" from the comparator has a hardware error. The type of error (stuck-at-1 or stuck-at-0) can be determined by inspecting the same-position bit in Register 1.

Figure 16(b) through (e) illustrate four examples of error scenarios. We explain how

PERFECTORY handles an error based on the examples. The first case we consider is a stuck-at-1 error. In fact, a stuck-at-1 error is benign and does not lead to a coherence problem in our framework–there will be potentially more invalidation messages than needed at invalidation events, but there is no harm to the correctness of the program execution. Figure 16(b) shows an example where a bit hit by a stuck-at-1 error is set (to have the value 1). In this case, PERFECTORY does not incur any further action at a later invalidation time than an error-free entry. Figure 16(c) gives another benign error example (failure to store "0" due to the hardware error) where PERFECTORY will conservatively generate one additional invalidation message to core 2. While a processor can continue operating correctly in the presence of a benign error, PERFECTORY may log this error to the operating system for other management purposes. Now, let's consider a stuck-at-0 error. A stuck-at-0 error should be recovered because it can lead to inadvertently dropping necessary invalidation messages, and hence, can cause incorrect program execution. Figure 16(d) shows a stuck-at-0 error that wouldn't allow writing "1" to the bit position. In this case, PERFECTORY detects the error and correctly generates two invalidation messages (to core 0 and 1). Figure 16(e) depicts the second case of a stuck-at-0 error. This time, the directory controller generates three invalidation messages to guarantee correct program execution to core 0, 1, and 2 (speculated). PERFECTORY detects that bit number 2 is faulty. However, because it is unable to decide whether the bit was previously set to 1 or 0, it speculatively generates an invalidation message for core 2.

The above examples show that PERFECTORY is capable of catching all hard errors that occur in a directory entry and their propagation is effectively prevented by conservatively generating invalidation messages. In a CMP that uses PERFECTORY, cores that receive an unnecessary invalidation message (generated speculatively on a detected error) simply drop the message when the specified cache block is not found locally. PERFECTORY generates at most k unnecessary invalidation messages when the directory entry has k errors. Surely, not all of these invalidation messages incurs performance penalty as we see in Figure 16(d). Note also that PERFECTORY does not use any additional storage space other than the two registers (Register 1 and Register 2 in Figure 16(a)) in the directory controller.

5.3 PROTECTING DIRECTORY ENTRIES FROM SOFT ERRORS

PERFECTORY's on-line error detection algorithm works well against hard errors and intermittent errors. However, it is not capable of detecting soft errors because they come from one-time events such as a hit by alpha particles. To offer immunity against soft errors, we propose to introduce a single parity bit in the sharer field and to detect errors at update times ("update time detection").

Imagine that a soft error occurred to a directory entry. At a later time, when the same directory entry is read, its parity bit will detect that a soft error has occurred since the last read of the same entry. The situation becomes complex if a hard error (due to aging, for example) occurred to the same directory entry. This unexpected hard error may interfere with the parity bit semantics. That is, a hard error that occurs after the last read time may look like a soft error according to the parity bit. To overcome this complication, we propose a novel way to update the parity bit whenever we detect a (new) hard error by further utilizing the sharer bits for the purpose of soft error immunity. We calculate and update the parity bit based on the adjusted sharer field that includes a hard error. If the parity bit itself is damaged from a new hardware error, the sharer field is modified to still guarantee soft error immunity. The goal of this coordinated sharer field and parity bit update algorithm is to tackle multiple hard errors and one soft error in a single directory entry.

Figure 17 illustrates how our hard error parity update is done, assuming an odd parity scheme. Figure 17(a) first describes the situation when a new hard error is detected. This case, however, does not require correcting the parity bit because the error has not damaged the original data. Hence, in this case, a soft error will be caught by the parity bit normally. Figure 17(b) presents a case where the new hard error flips a bit in the original data. Therefore, the parity bit indicates an error condition even though the error didn't come from a soft error. In this case, PERFECTORY needs to update the parity bit (from 1 to 0) once it identifies the hard error. When the controller accesses this block later, it can detect a soft error using the parity bit. Once a soft error is detected and invalidation is needed, the controller sends invalidation messages to all cores. Finally, although some of them are speculative invalidation messages, they do not affect program correctness. They

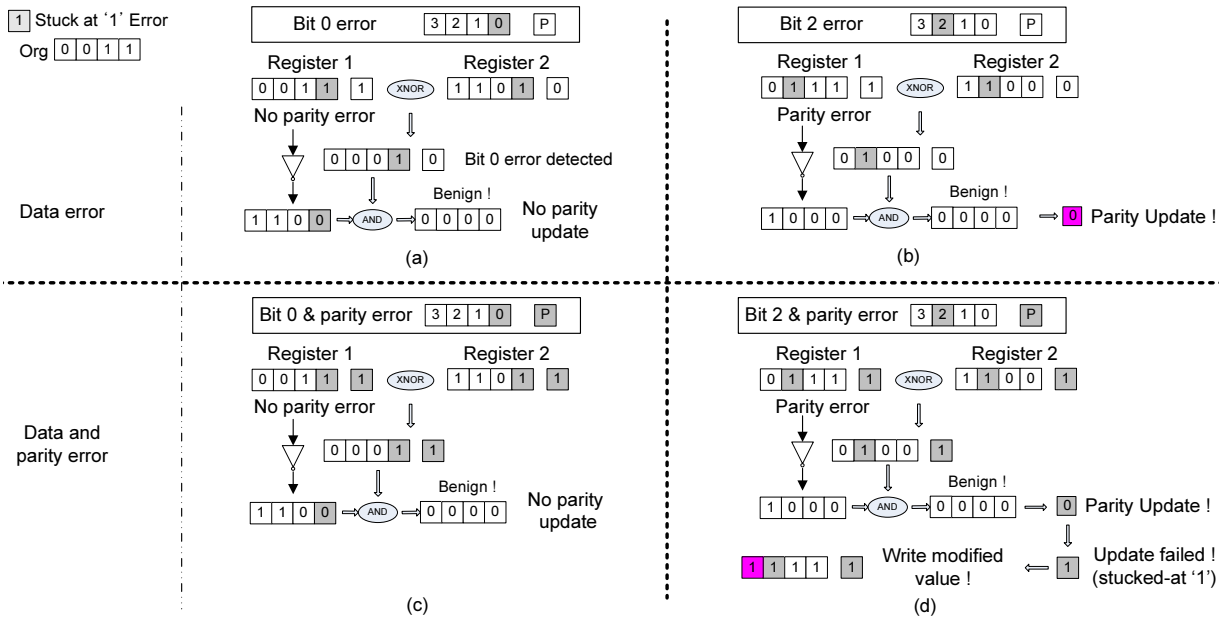


Figure 17: Soft error protection examples using the update-time detection. (a) Undetectable intermittent fault; parity is reserved. (b) 1 bit hard/intermittent fault detected at write time; parity is updated. (c) Benign multi-bit faults including parity. (d) Parity fault is detected, parity update fails, and error compensation is done by updating the data value.

have negligible performance impact because soft errors are rare.

Figure 17(c) and (d) show situations when the parity bit is damaged. In Figure 17(c), parity update is unnecessary. Although the parity bit has a hard error, the original parity value is the same as the error value (1 in this case) and the fault in the parity bit does not affect the soft error detectability. However, Figure 17(d) is the case where we are unable to update parity because of the damaged parity bit. In this example, to make parity odd, we replace a "0" with a "1" in bit 3 (forcing core 3 into becoming an "artificial" sharer). Therefore, two additional invalidation messages can be generated later: One from the hard error and the other from the parity compensation. If there is a soft error later, PERFECTORY can detect it using the odd parity.

Although PERFECTORY has hard, intermittent, and soft error immunity, it has a limitation. If multiple errors occur in a single directory entry after the last access and at least one of them is a soft error, PERFECTORY is unable to detect them all. For example,

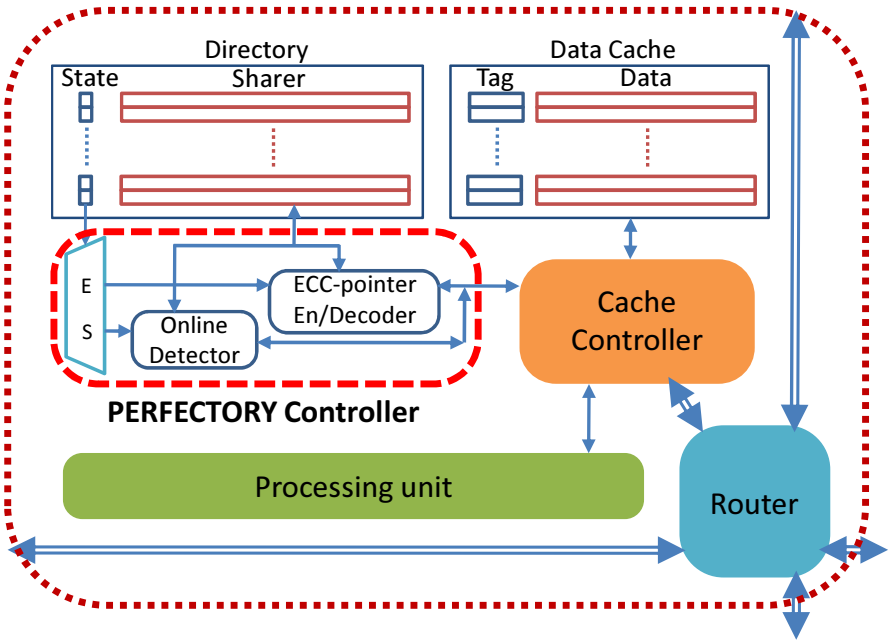


Figure 18: PERFECTORY controller architecture.

one harmful hard error and one soft error occur after the last access of the directory entry. Because PERFECTORY primarily focuses on hard/intermittent errors and this multi-bit error case is extremely rare, we do not consider this case in this paper. We note that existing ECC schemes are also unable to handle this case. We also note that there are design practices that prevent multiple soft errors by interleaving bit lines [70, 59].

5.4 PERFECTORY CONTROLLER ARCHITECTURE

Figure 18 depicts the hardware architecture of the PERFECTORY controller. It consists of two parts: ECC encoder/decoder for the Exclusive state and on-line error detector for the Shared state. The output of these two blocks goes to the cache controller with the recovered sharer information. When the state is Shared, the delivered sharer information may include speculation. The ECC-pointer en/decoder block has multiple en/decoders and an arbiter. In a 64-core directory, maximum five ECC-pointer en/decoders can be embedded. When the directory entry is updated to assume the state Exclusive, the ECC encoder converts the

Table 8: Simulated systems and workload parameters.

Parameter	
Core	In-order core, 2GHz, 64 cores
L1 cache	16kB split I/D, 4-way assoc, 64B lines, 1-cycle hit
L2 cache	256kB per core, 16-way assoc, 64B lines, 7-cycle hit
On-chip network	8×8 mesh network, 1-cycle queue latency, 2-cycle routing latency, 16 entry queue for both ingress and egress
Memory	400-cycle initial latency, 10-cycle latency in the DRAM row buffer, 4 DRAM controllers (each corner)

Workload	Problem Size
Barnes	65,536 particles
Cholesky	Tk29.O
Lu	1024×1024 matrix,64×64 block
Ocean	514×514 grid

core number into the ECC-pointer format. When the directory entry is read, the five ECC decoders decode the five ECC-pointer pairs in the sharer field in parallel and produce results. If there is any uncorrectable error among five outputs, that output value is discarded. If more than one ECC-pointer pair has uncorrectable errors, the arbiter compares the five outputs and selects the output. In an extreme case, if all five decoders are unable to recover from errors, that directory entry is disabled. The online detector, whose architecture is shown in Figure 16(a), is connected between the cache controller and the directory memory.

5.5 EVALUATION

This section evaluates the effectiveness of PERFECTORY and other memory fault covering schemes in protecting the coherence directory. We use yield and life-time reliability as two key metrics. When comparing PERFECTORY with graceful degradation schemes, we also use performance and hardware cost (chip area) as metrics.

5.5.1 Experimental setup

We use a 64-core CMP model similar to the one in Figure 2.3(a) for all evaluations. It has 64 tiles each with a two-issue in-order core, linked by a 8-by-8 mesh network. Each core has

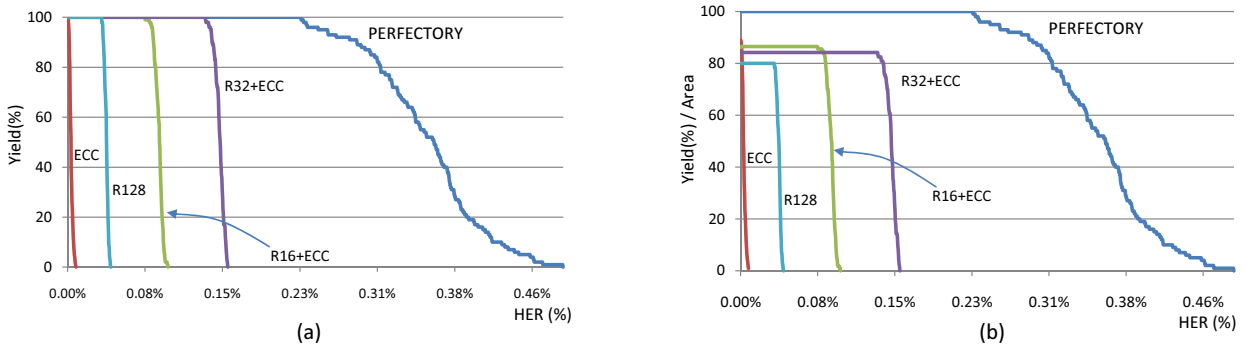


Figure 19: (a) Yield vs. HER. R32 is 32 redundant blocks per way (i.e., 512 redundant blocks for 16 way). R32+ECC and R16+ECC are ECC with 32 or 16 redundant blocks. (b) Yield/Area vs. HER.

16kB 4-way L1 I/D caches and a 256kB 16-way L2 cache slice which collectively constructs a 16MB globally shared L2 cache.

We compare four yield enhancement schemes in our experiments: Circuit-level redundancy, ECC without redundancy, ECC with redundancy, and PERFECTORY. To evaluate their yield, we first calculate the directory memory area under the four schemes. We then inject hard errors into the directory memory area using a uniformly random distribution. Based on the errors injected, we evaluate if all the errors can be covered by the yield enhancement schemes. Our evaluation methodology uses 100 Monte Carlo simulation runs per error injection level.

With regard to resilience to run-time errors (soft error), we compare three schemes in this section: ECC, ECC with disabling, and PERFECTORY, using MTTF (mean time to failure). ECC with disabling is the combination of ECC and a state-of-the-art memory block disabling scheme similar to the Intel Cache Safe Technology [17]. When ECC is unable to correct an error (e.g., a two-bit error), the ECC-disabling scheme discards the erroneous block and does not further use it. We assume the maximum number of disabled blocks is eight for the ECC-disabling scheme. In this experiment, our machine model is a 100-processor cluster where each processor is a 64-core CMP. We assume a run-time error rate of 1,000 FIT per megabit [66] and measure how long the modeled coherence directory can tolerate run-time errors given a hard error injection level.

When evaluating PERFECTORY and disabling schemes for performance degradation, we use a cycle-accurate trace-driven CMP architecture simulator [18]. In our simulation, contention in DRAM memory ports and network switches is modeled, including network queuing delay, network link delay, and delays caused by network congestion. We use four SPLASH-2 parallel benchmarks [86] to construct workloads. Each application runs with 16 threads. To fully exercise our 64-core CMP machine model, we form 64-thread workloads by either running together four copies of a benchmark program ("homogeneous") or running four different programs ("heterogeneous"). Hence, we have four homogeneous workloads (from four benchmark programs) and one heterogeneous workload with all four benchmark programs. Each program runs in one of four geometric groups with 16 cores: NW, NE, SE, and SW. Simulation and workload parameters are summarized in Table 8. For presentation, we selected three HER values: Light (0.05%), medium (1%), and heavy (10%). The hard error ratio (HER) is simply the ratio between the number of injected errors and the total number of available memory bits.

5.5.2 Result

5.5.2.1 Yield Figure 19(a) presents the yields for four yield enhancing techniques: Redundancy, ECC, ECC+redundancy, and PERFECTORY. Note that we use one very large redundancy only configuration (i.e., R128 which has 128 redundant rows), and two ECC configurations with moderate redundancy (i.e., R16+ECC and R32+ECC. They have 16 and 32 redundant rows, respectively). The result reveals that PERFECTORY achieves higher yield than other schemes by a large margin, especially when there are many errors. At 0.2% HER, all other schemes except PERFECTORY have 0% yield, whereas PERFECTORY's yield is still 100%. It is clear that the yield of circuit-level redundancy and ECC falls quickly as HER is increased. The redundancy schemes tolerate errors only up to the predetermined redundant row count. ECC endures only one bit error for each block. As HER is increased and directory entries begin to have multiple errors, ECC fails quickly. Although the combination of ECC and circuit-level redundancy schemes has much improved resilience against errors compared with either of them alone, their effectiveness drops much earlier

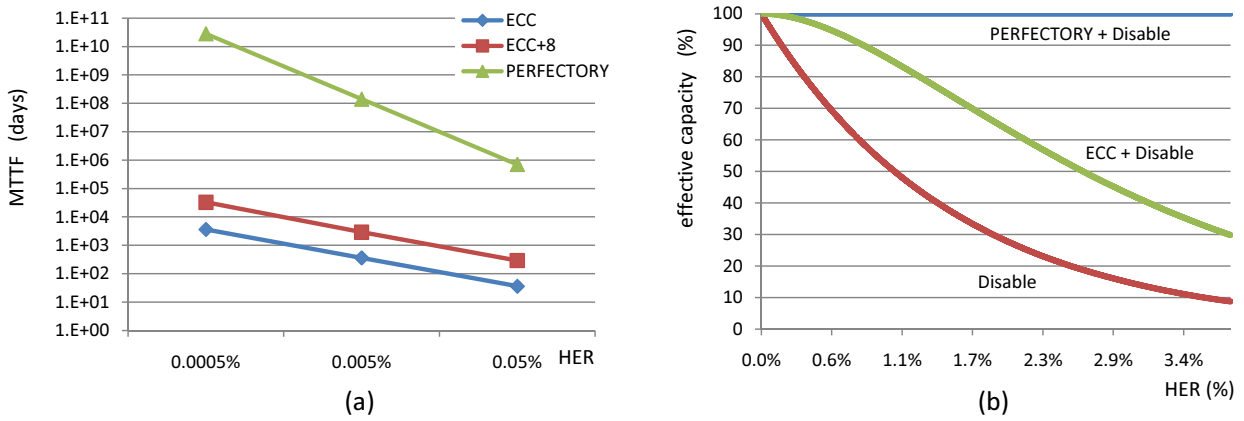


Figure 20: (a) MTTF (days) vs. HER. ECC only fails at the first multi-bit error. ECC+8 can disable up to eight blocks but fails at the ninth multi-bit error. PERFECTORY fails at the first block that it is unable to correct (Section 5.2.1). (b) Effective capacity vs. HER. ECC+Disable combines ECC with unlimited disabling. Disable is an unlimited disabling scheme.

than PERFECTORY.

As previously discussed, PERFECTORY does not use additional storage space while circuit-level redundancy and ECC schemes require additional silicon area. Note that both yield and silicon area affect the cost of a chip. To expose the area overhead of each scheme in the yield calculation, Figure 19(b) compares the same set of schemes using a slightly different metric, *yield per area*. The result illustrates the higher cost-effectiveness of PERFECTORY than other schemes even more clearly.

5.5.2.2 Resilience to run-time errors To illustrate run-time error resiliency, MTTF for 100 processors is shown in Figure 20(a). The result shows that ECC does not provide high resilience against run-time errors. MTTF for ECC at 0.05% HER is more or less one month (36 days). This low MTTF is from the fact that ECC does not provide soft error immunity for the blocks that were already struck by a hard error. As HER is increased, many directory entries become vulnerable to soft errors. ECC with disabling improves error resilience, but not by a large margin. With 0.0005% HER, ECC with disabling has a relatively high MTTF (89 years). However, at 0.05% HER, its MTTF falls quickly to less than one year (288 days). PERFECTORY shows a much longer MTTF than the other schemes–1,934 years at

0.05% HER. The resilience of PERFECTORY against soft errors comes from its capability to detect a soft error in the presence of multiple hard errors with online testing.

The previous result showed that disabling adds to MTTF. Naturally, one may want to implement an "unlimited" disabling scheme to extend MTTF further. The key issue in unlimited graceful degradation is the performance loss as more resources are deleted. Hence, we address the question of "how many directory entries remain operational as more and more hard errors are injected under an unlimited disabling scheme?" before we actually study the performance and area cost of PERFECTORY and an unlimited disabling scheme. One can think of hard errors in this experiment as errors caused by aging [67]. Figure 20(b) presents the result–the effective capacity of the coherence directory (ratio of usable entries to all entries) as HER is varied. We compare three schemes: Disable (unlimited disabling), ECC+Disable (unlimited disabling with ECC), and PERFECTORY+Disable (unlimited disabling with PERFECTORY).

The result shows that PERFECTORY+Disable loses the smallest amount of effective directory capacity as HER is increased. At nearly 3% HER, PERFECTORY+Disable can still use more than 99.9% of the total capacity. Disable and ECC+Disable get only 14.7% and 42.6% of the total capacity, respectively, in this heavy damage situation. The result indicates that PERFECTORY is capable of significantly improving a chip's life-time reliability when augmented with a disabling capability.

5.5.2.3 Chip area overhead PERFECTORY's hardware overhead has two parts: controller logic in Figure 18 and memory elements (i.e., a parity bit per directory entry). For the controller logic overhead, state selector, online detector, and ECC-pointer en/decoder should be considered. Figure 21 illustrates the area overhead for PERFECTORY, ECC without redundancy, and ECC with redundancy. ECC has 12.5% fixed area overhead from (72,64) SECDED code. Therefore, nearly 15% area overhead is required for ECC even without redundancy. However, the area overhead for PERFECTORY is less than 4% for more than 64 cores and decreases as the number of cores grows.

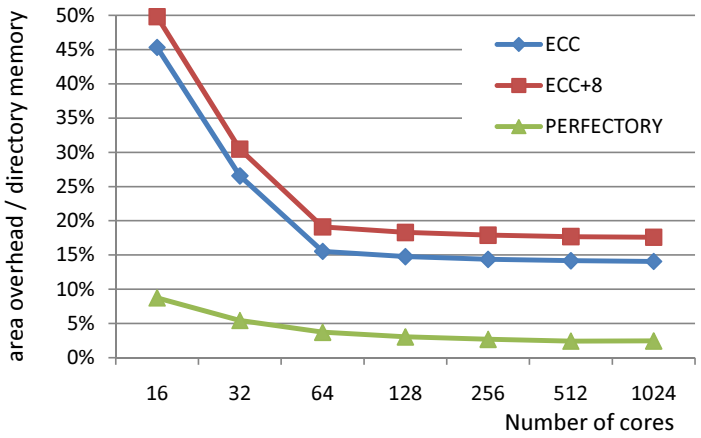


Figure 21: Area overhead. For the overhead of ECC and ECC+8, memory for ECC and En/Decoder logic are considered. (72,64) SECDED code is used for more than 64 cores. For PERFECTORY, state selector, multiple ECC En/Decoder, online-detector, and one bit parity are considered. To calculate the hardware implementation cost of ECC En/Decoder, Hsiao's SECDED code implementation is used [31]. 4 PMOS and 4 NMOS transistors are assumed to calculate one XOR gate area [30].

5.5.2.4 Performance overhead Our performance overhead evaluation looks at two aspects: low-level circuit latency and high-level program performance.

In Figure 18, there are two circuit paths between the directory memory and the cache controller: one for E state and the other for S state. State selector of the PERFECTORY controller in Figure 18 can be implemented with inverters. Either E state or S state has at most one inverter delay from the state selector. HSPICE simulation shows that it has less than 0.2ns latency in 65nm technology with 1V supply voltage. Smaller technologies have even less latency. The online detector has much larger latency than the ECC-pointer En/decoder in Figure 18. Therefore, if the combined latency of the online detector and state selector's circuit is less than the data cache access latency, PERFECTORY can operate without any circuit level performance penalty. Table 9 compares PERFECTORY's circuit latency with the data cache access latency. PERFECTORY operation is assumed to access the directory memory three times (i.e., read, write, and read). For all technologies, PERFECTORY's operation is less than L2 cache access latency.

For high-level performance simulation, we randomly generated faults according to HER, and inject them into the directory memory area in the modeled CMP chip. Each error is

Table 9: PERFECTORY's circuit performance overhead. CACTI [80] is used to estimate access latencies. L2 cache model follows Table 8. SRAM only model is assumed for the directory memory. 1.0V supply voltage is assumed.

Technology	L2 access latency	PERFECTORY operation
65nm	3.4570ns	2.0307ns
45nm	2.2885ns	1.1886ns
32nm	1.7538ns	0.8925ns

either stuck-at-1 or stuck-at-0. We compare three schemes in this section: Disable (unlimited disabling), ECC+Disable (ECC with unlimited disabling), and PERFECTORY+Disable (PERFECTORY with unlimited disabling).

Figure 22(a)–(c) present the number of misses, the number of invalidation misses, and program slowdowns for the homogeneous workload under the three schemes. Similarly, Figure 23 gives the program slowdowns for the heterogeneous workload. While our heavy error scenario (10% HER) is unlikely in today's processor chips, Intel suggested the possibility of an extremely unreliable chip design environment where 20% of all on-chip devices are unreliable [12]. When a chip's supply voltage is lowered to save energy, a large chunk of memory devices may become temporarily unreliable [85]. Therefore, our study using the heavy error scenario is a meaningful limit study that evaluates our proposed and other existing schemes in terms of their error resiliency under extreme conditions.

Additional L2 cache misses in Figure 22(a) come from disabled cache blocks due to hard errors. Disable and ECC+Disable schemes have significantly more cache misses than PERFECTORY. PERFECTORY has negligible additional cache misses for the light and medium error conditions (0.05% and 1% HER). In the heavy error condition (10% HER), PERFECTORY limits the additional misses to 62.6% whereas other two schemes have more than 800 times the original misses.

Additional invalidation messages in Figure 22(b) come from two sources: Reduced cache capacity and speculative invalidations. Speculative invalidations make PERFECTORY + Disable have more invalidation messages than other two schemes. However, PERFECTORY

has significant less invalidation messages for OCEAN with 1% HER. In this case, reduced cache capacity of Disable and ECC+Disable schemes results in more cache blocks being replaced at the bottom of LRU stacks, incurring invalidation messages. Interesting results are shown for 10% HER. Disable and ECC+Disable have almost no invalidation messages (indicated by -100%). If there are few available directory blocks for the heavy error condition, accesses miss in the L2 caches and go to the main memory. In this case, there is no invalidation messages because the directory does not manage any coherence informations. However, PERFECTORY has huge additional invalidation messages because it recovers faulty blocks and generates speculative invalidation messages.

Program slowdowns in Figure 22(c) and 23 are a result of additional L2 cache misses and additional invalidation messages. L2 misses increase not only average L2 cache access time but also network traffic because memory access traffic consumes on-chip network bandwidth. They may also increase memory access time because of heavier contention at the memory ports. Invalidation messages also add to the on-chip network traffic and potentially increase network contention.

Our results show that under the heavy error scenario, PERFECTORY limits program performance degradation to 1% for the homogeneous configuration and 16.9% for the heterogeneous configuration, at 10% HER. The Disable and ECC+Disable schemes, however, suffer from significant performance degradation: On average more than 10 times and 3.5 times for the homogeneous and heterogeneous configuration, respectively. A large amount of additional cache misses and invalidation messages cause this performance degradation. Even under a heavy error scenario, PERFECTORY was able to limit the overhead of additional, speculative invalidations. PERFECTORY generates speculative invalidation messages only when there are errors and the original sharer information has discrepancy with the error pattern. Note also that invalidation messages are generated only when there is a state transition from Shared to other states, such as Exclusive and Invalid. Three benchmarks–Barnes, LU, and Ocean–out of four have few state transitions from Shared to other states [20]. This program behavior limits the chance of program performance degradation due to speculative invalidation messages. Lastly, although the Disable and ECC+Disable schemes do not generate speculative invalidation messages, their reduced cache capacity leads to not only

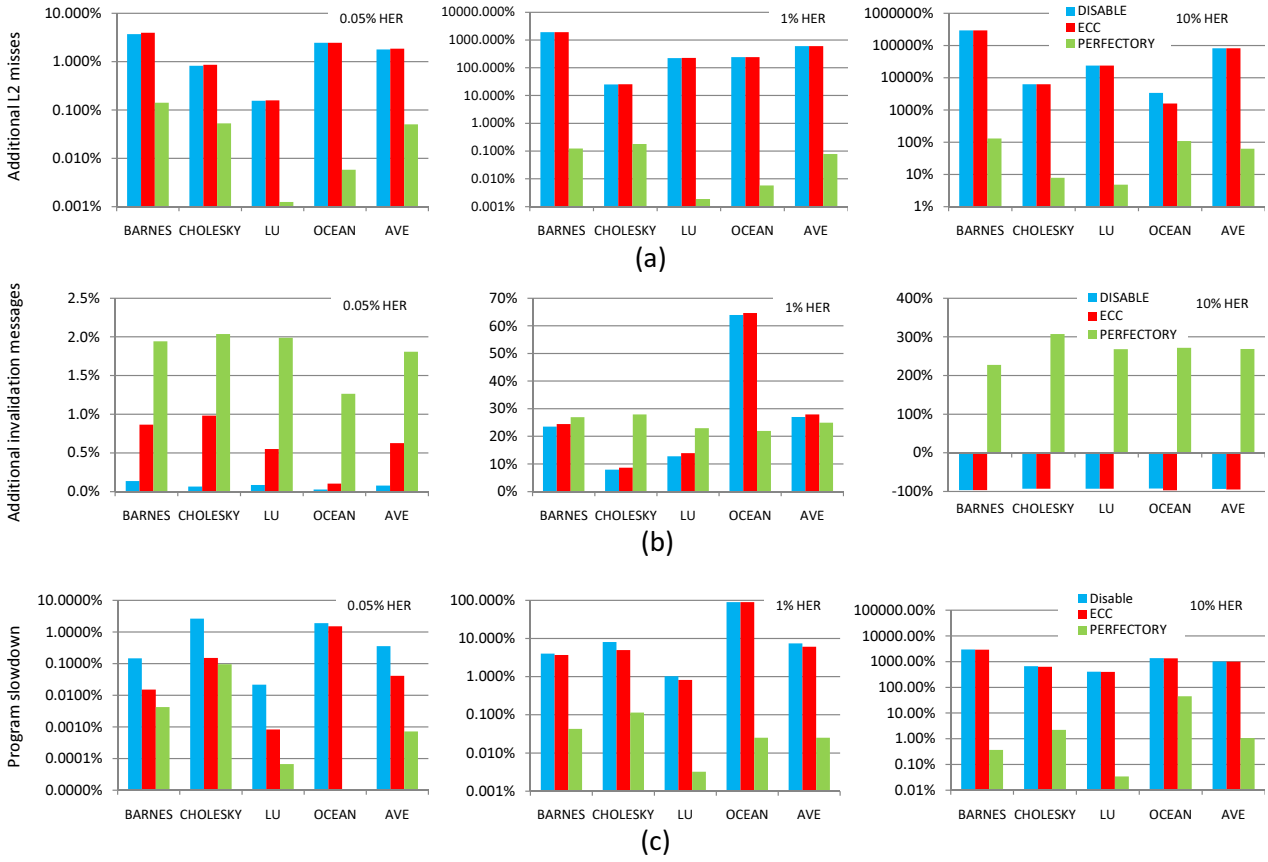


Figure 22: Homogeneous workload: (a) Additional L2 misses. (b) Additional invalidation messages. (c) Program slowdown. All results are normalized to the "no error" configuration.

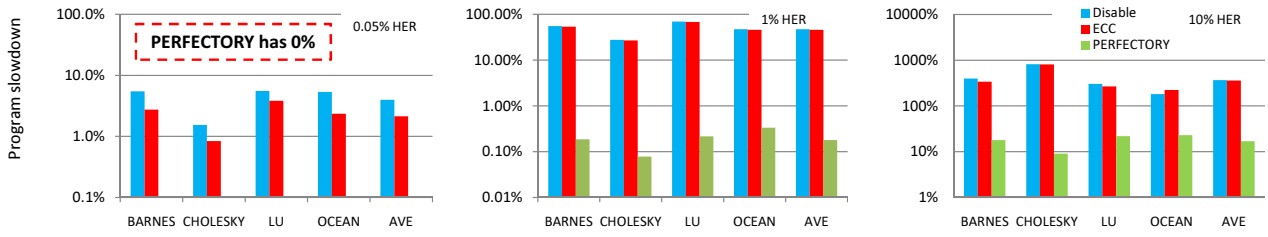


Figure 23: Heterogeneous workload. Individual program's slowdown is normalized to the "no error" configuration.

additional cache misses but also additional invalidation messages. Eventually, they lead to larger performance degradation than in PERFECTORY+Disable.

5.6 SUMMARY

The core count in CMPs is increasing fast and so is on-chip cache memory capacity. The on-chip coherence directory grows in size as core count and cache capacity increase. Considering that chips can have more frequent fault occurrences with future technologies, it becomes imperative to provide robust mechanisms to protect the on-chip coherence directory memory for reliable computing.

This chapter proposed PERFECTORY, a comprehensive, low-overhead microarchitectural scheme to detect and correct hardware errors that occur in the coherence directory memory. Evaluation results show that PERFECTORY achieves very high chip yield–at 0.2% HER, yield is 100% with PERFECTORY, whereas a 32-way row redandancy scheme's yield is virtually zero. Furthermore, PERFECTORY significantly improves the life-time reliability of the directory memory. The MTTF using a 100-processor cluster configuration is 1,934 years with PERFECTORY at 0.05% HER and 1,000 FIT, whereas ECC's MTTF is only 36 days under the same condition. Compared with existing yield improvement strategies and soft error protection mechanisms, PERFECTORY has a substantially smaller chip area cost and performance slowdown. It requires only one bit per directory entry. Finally, PERFECTORY shows less than 0.1% program performance degradation at 0.05% HER whereas a state-of-the-art disabling scheme has 4% program performance degradation.

6.0 ADDRESSING YIELD FOR L2 CACHE

6.1 OVERVIEW OF STIMULUSCACHE

Given the high likelihood of available excess caches (see Section 3.6), one would naturally want to utilize them to improve system performance. A naïve strategy could simply allocate excess caches to cores that run cache capacity-hungry applications. Adding more capacity to specific cores creates *virtual L2 caches* which have more capacity than other caches. However, with diverse workloads on multiple virtual machines (VMs), deriving a good excess cache allocation can become complex. For example, the user might pursue the best performance, while, in another case, the user may want to guarantee QoS and fairness of specific applications. To achieve these potential goals, we propose a hardware/software cooperative design approach. In this section, we illustrate the proposed StimulusCache framework by discussing its hardware support, software support, and an extended example.

6.1.1 Hardware design support

Shared and private caches are two common L2 cache designs. There are also many hybrid (or adaptive) schemes [16, 72, 71]. A private L2 cache design has several benefits over a shared L2 cache design: fast access latency, simple design, resource/performance isolation, and less network traffic overhead. Such a private design typically has poor utilization. However, the extra cache capacity from available excess caches can mitigate this problem. Thus, our initial StimulusCache design is based on the private L2 cache architecture such as IBM's Power6 [32] and AMD's Phenom [5].

Figure 24(a) shows the fault isolation point of a conventional core disabling technique and

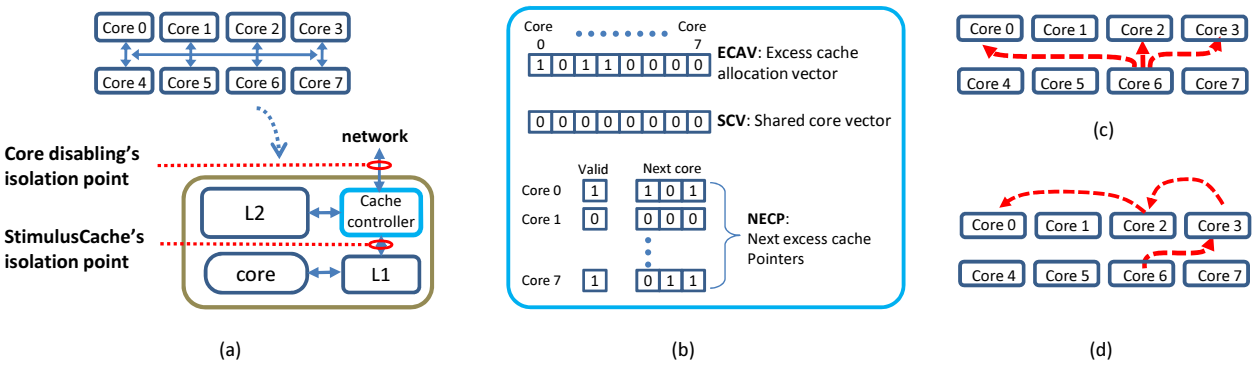


Figure 24: (a) Fault isolation point comparison: core disabling and StimulusCache. (b) New data structure in StimulusCache's cache controller: excess cache allocation vector (ECAV), shared core vector (SCV), and next excess cache pointers (NECP). ECAV shows which excess caches have been allocated to this functional core. A faulty core's ECAV is empty. SCV lists the cores that use this excess cache. A functional core has zero value. NECP shows the next level excess cache to search on a miss. A working example is shown in Figure 25. (c) Parallel search using ECAV. (d) Serial search using NECP.

StimulusCache in an 8-core CMP that has a private L2 cache per core. Wherever faults occur in the processing logic, conventional core disabling takes offline the whole core including its private L2 cache. Thus, the fault isolation point is the core's network interface. StimulusCache aggressively pushes the isolation point beyond the L2 cache controller. Consequently, StimulusCache can salvage the L2 cache as long as the L2 cache and cache controller are fault-free.

In StimulusCache, each core should be able to access excess cache without limitation. We introduce a set of hardware data structures in the cache controllers, as shown in Figure 24(b), to provide flexible accessibility to excess cache. The excess cache allocation vector (ECAV) shows which caches should be examined to find requested data on a local L2 miss. Using ECAV, multiple excess caches can be accessed in parallel as shown in Figure 24(c). The Shared Core Vector (SVC) assists cache coherence and will be discussed in detail below. Lastly, Next Excess Cache Pointers (NECP) enable fine-grained excess cache management. Each pointer points to the next memory entity to be accessed, which is either another excess cache or main memory. NECP forms access chains of excess caches for individual cores as shown in Figure 24(d). With ECAV and NECP, StimulusCache supports both parallel and

sequential search of the excess caches. Parallel access is faster while sequential access has less network traffic and power consumption. The best choice could be determined by the overall system management goal; for example, performance or power optimization. Memory overhead from the data structures is $(3N + N \log_2 N)$ bits per core for an N-core CMP; the overhead corresponds to 6 bytes for an 8-core CMP and 32 bytes for a 32-core CMP, or only 0.001% and 0.006% of a 512kB L2 cache in term of bit counts.

Although excess caches can be used to improve performance, static allocation of excessable for entire program execution may not exploit the full potential of excess cache because programs have different phases with varying memory demand. To support program phase adaptability, excess caches should be dynamically allocated to cores based on performance monitoring. StimulusCache's advantage for dynamic allocation is its inherent performance monitoring capability at cache bank granularity. For example, data flow-in, access, hit and miss counts, which are already implemented in CMPs [4], can be measured and used to fully utilize the potential of excess caches.

Coherence management in StimulusCache is similar to a private L2 cache. For moderate scales (up to 8 cores), broadcast is used for cache coherence. For large scale (greater than 8 cores), a directory-based scheme is used [16, 50]. However, to utilize excess caches, the coherence protocol has to be changed. An excess cache can be shared by multiple cores, or it can be exclusively allocated to a specific core. To manage cache coherency, the cache controller has SCV as shown in Figure 24(b). The SCV for a faulty core lists the functional cores that utilize the excess cache of the faulty core. When L1 data invalidation occurs, the SCV identifies the cores that need to receive an invalidation message. For functional cores, SCV entries are empty because their local L2 caches are not shared.

6.1.2 Software support

An excess cache is a shared resource among multiple cores; system software (e.g., the OS or VMM) has to decide how to allocate the available excess caches. The system software should assign an excess cache to a core in a way that meets application needs by properly setting the values of ECAV, SCV, and NECP in the cache controllers.

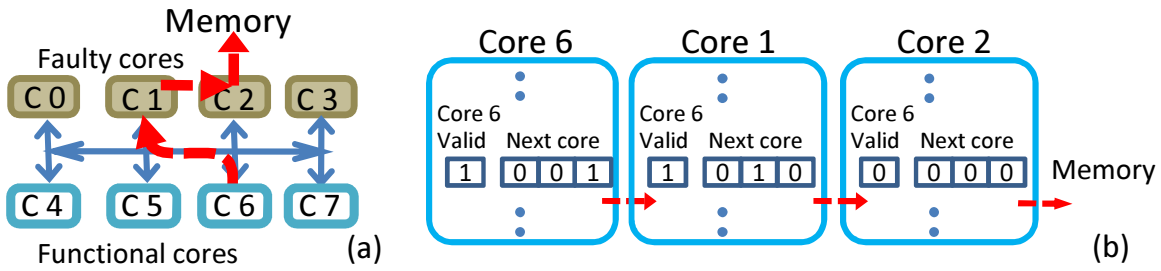


Figure 25: Excess cache allocation example. (a) Excess caches from four faulty cores (core 0, 1, 2, and 3). (b) NECP in core 6, 1, and 2. An excess cache access chain for core 6 is shown. A zero valid bit indicates that the excess cache is the last one in the chain before the main memory. The access sequence is, therefore, core 6 (working core) ⇒ core 1 (excess cache) ⇒ core 2 (excess cache) ⇒ main memory.

Depending on the resource utilization policy, the system software decides if an excess cache should be exclusively allocated to a core. Exclusive allocation guarantees performance isolation of each core. However, if there is no information about memory demands, a fixed exclusive allocation is somewhat arbitrary. In that case, evenly allocating available excess caches to all sound cores is a reasonable choice. If there is not enough excess cache for all available cores, the excess caches are allocated to some cores, and the OS can schedule memory-intensive workloads to the cores with excess cache. Shared allocation can exploit the full potential of excess cache usage. However, the excess caches could be unfairly shared if some cores are running cache capacity thrashing programs.

6.1.3 An extended example

Figure 25 gives an example that shows how excess caches can be allocated. In this example, cores 0 to 3 are faulty, and thus, they provide excess caches. Cores 4 to 7 are functional. Core 6 has been allocated two excess caches from core 1 and core 2. The excess cache from core 1 has higher priority (it is at the first of an access chain). For accessing excess cache to examine data (i.e., a data read), core 6 can search the excess caches of core 1 and 2 simultaneously in parallel or sequentially as shown.

When data is written to the excess cache (e.g., a data eviction from the local L2 cache of core 6), the destination cache of the data has to be determined. Figure 25(a) shows an

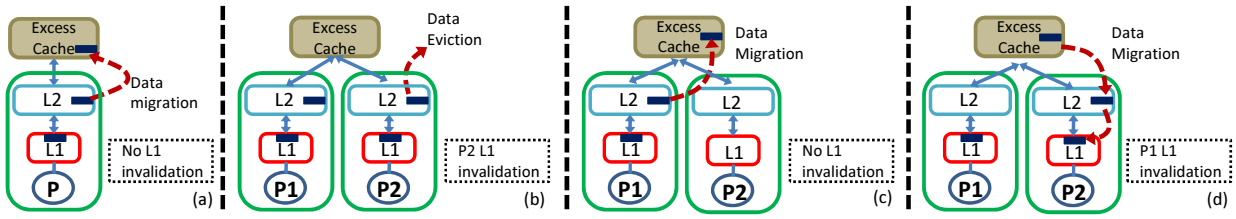


Figure 26: StimulusCache coherency examples. (a) No L1 invalidation for data migration from L2 to exclusive excess cache. (b) L1 invalidation for data eviction. Data migration does not occur because P2 has the same data. (c) No L1 invalidation for data migration from L2 to shared excess cache. If no other core has the same data like (b), no L1 invalidation is needed because only P1 has valid L1 data. (d) L1 invalidation for data migration. If P2 migrates the data from the shared excess cache to the local L2 cache, P1's L1 data should be invalidated.

excess cache access chain. In this example, core 6's L2 eviction data goes to the excess cache in core 1, identified with the NECP (in core 6). If the data should be written to the next cache in the chain, it goes to the excess cache in core 2 based on the NECP in core 1. Figure 25(b) shows how the NECPs in the cache controllers are used to build the excess cache access chain for Figure 25(a).

Figure 26 shows example scenarios of how cache coherence is done in StimulusCache. The excess caches for core along with the core's private L2 cache, create *virtual L2 domain*. Each core has valid *inclusiveness* if the data in L1 cache has the same copy in the virtual L2 domain. Therefore, if exclusive L2 data is migrated to an excess cache, an L1 data invalidation is not needed. As shown in Figure 26(a), only one copy of valid data is kept in either the L2 cache or the excess cache. Figure 26(b) shows a different scenario where two cores have the same data (i.e., each has a replica of the data) which is shared by cache-to-cache transfer. If one core should evict this shared data, the data is not migrated to the excess cache. Instead, it is simply evicted as there is a valid copy in P2's L2 cache, satisfying the L2-L1 inclusiveness requirement. Figure 26(c) shows another scenario. In this case, if exclusive data in L2 is migrated to the excess cache, no L1 invalidation is needed because there is only one L1 data. Finally, Figure 26(d) depicts data migration that incurs L1 invalidation. To maintain L2-L1 inclusiveness, if the data in the excess cache is migrated to P2's L2 cache, then P1's L1 data should be invalidated. The proposed hardware support provides sufficient information to achieve coherency with excess caches.

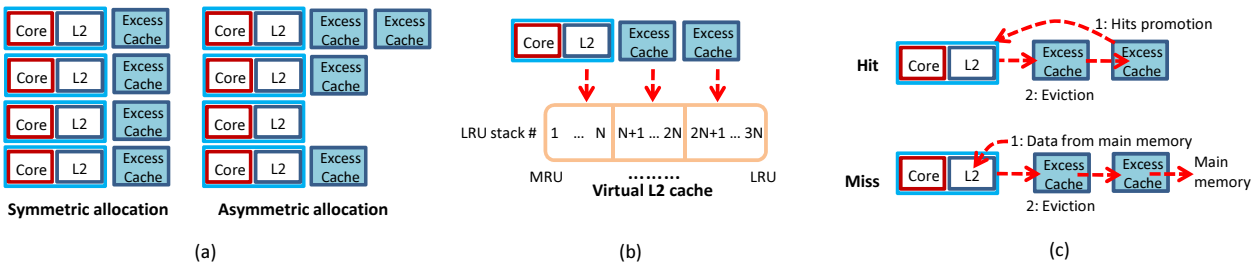


Figure 27: Static private scheme. (a) Two example allocations, symmetric and asymmetric. (b) $3N$-way virtual L2 cache with two N-way excess caches. (c) Data propagation in an excess cache chain during excess cache hit and miss. On a hit, (1) hit data is promoted from the hit excess cache to the local L2 cache; and (2) a block may be replaced from the local cache and propagated to the head of the excess cache chain, and so on. The propagation of a block may extend to the excess cache that previously hit the most, as it has space from promoting a hit block. On a miss, (1) data from the main memory is brought to the local L2; (2) a replaced block causes a cascading propagation from the local L2 cache through the excess cache chain; and (3) a block from the tail of the excess cache chain may move to main memory.

6.2 EXCESS CACHE UTILIZATION POLICIES

Based on the hardware and software support described in Section 6.1, we present in this section three policies to exploit excess caches.

6.2.1 Static private: static partition, private cache

This scheme exclusively allocates the available excess caches to individual cores: only one core can use a particular excess cache as assigned by the system software. Figure 27(a) shows two examples of a static allocation of excess caches to cores. If the workload on multiple cores have similar memory demands, the available excess caches can be uniformly assigned to cores (symmetric case). A server workload or a well-balanced multithreaded workload are good examples of this case. However, if a workload has particularly high memory demands, then more excess caches can be assigned to a specific core for the workload. This configuration naturally generates an asymmetric CMP as shown in Figure 27(a).

In effect, the static private scheme expands a core's L2 cache associativity to $(K+1)N$ using K excess caches that are N-way associative. Figure 27(b) shows this property. When

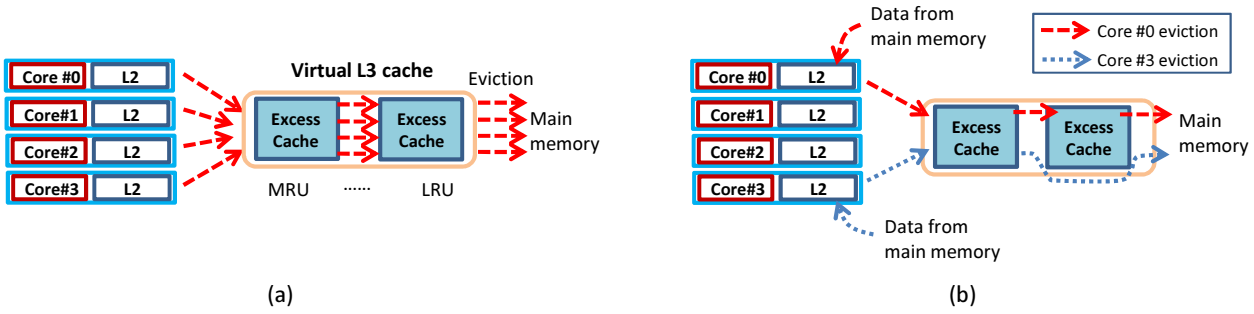


Figure 28: Static sharing scheme. (a) Homogeneous static sharing. (b) Heterogeneous static sharing.

data is found in a local L2 cache, the local L2 cache provides the data. If the data is not found in the local L2 cache (L2 cache miss), the assigned excess caches are searched to find the data. Because the same index bits are used during the search through multiple caches, each set's associativity is effectively increased. Figure 27(c) shows the two cases where data propagation from/to excess caches is needed. As a block would gradually move to the LRU position with the introduction of a new cache block to the same set, a block evicted from the local cache is entered into the next excess cache in the access chain.

6.2.2 Static sharing: static allocation, shared cache

Workloads may not have memory demand that matches cache bank granularity. For example, one workload may need half of the available excess cache capacity while another workload may need a little more capacity than one excess cache. With the static private scheme, some cores may waste excess cache capacity while other cores could use more. In this case, more performance could be extracted if the available excess caches are shared between workloads to fully exploit the available excess cache capacity. The static sharing scheme uses the available excess caches as a shared resource for multiple cores as shown in Figure 28(a). The basic operation of the static sharing scheme is similar to the static private scheme except that the excess caches are accessible to *all* assigned cores. If applications on the cores have balanced memory demands, this scheme can maximize the total throughput. The excess caches can also be allocated "unevenly" to an application with a high demand. If other applications

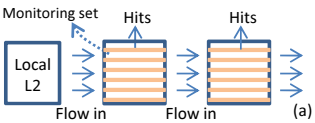
secure large benefits from not sharing with specific applications (i.e., due to interference), such an uneven allocation may prove desirable. Figure 28(b) shows an example in which core 3 has limited access to the excess cache. Core 0 can access two excess caches while core 3 can access only one excess cache.

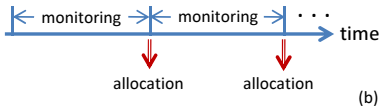
The static sharing scheme can be particularly effective for shared-memory multithreaded workloads because shared data do not have to be replicated in the excess caches (unlike the static private scheme). Furthermore, "balanced" multithreaded workloads typically have similar memory demands from multiple threads. In this case, the excess caches can be effectively shared by multiple threads in one application. If the initialization thread of a multithreaded workload heavily uses memory, then the static sharing scheme will work like the static private scheme because no other threads usually need cache capacity in the initialization phase.

6.2.3 Dynamic sharing: dynamic partition, shared cache

Static sharing has two potential limitations. It does not adapt to workload phase behavior, nor does it prevent wasteful usage of the excess cache capacity by an application that thrashes. While "capacity thrashing" applications do not benefit from excess caches, they can limit other applications' potential benefits. To overcome these limitations, we propose a dynamic sharing scheme where cache capacity demands from cores are continuously monitored and excess caches are allocated dynamically to maximize their utility.

Figure 29 illustrates how the dynamic sharing scheme operates. We employ "cache monitoring sets" (Figure 29(a)) that collect two key pieces of information, *flow-in counts* and *hit counts*. The counters at the monitoring sets count cache flow-ins and cache hits continuously during a "monitoring period" and are reset as the period expires and a new period starts. At the end of each monitoring period, a new excess cache allocation to use in the next period is determined based on the information collected during the current monitoring period (Figure 29(b)). We empirically find that 1M cycle period is good enough to determine excess cache allocation. The monitoring sets are accessed by all participating cores, while other non-monitoring sets are accessed by only the allocated cores. We find that





H/F = (Hits/Flow-in)	Action
H/F = 0	Counter −−
0 < H/F ≤ Mth	No action
Mth < H/F ≤ Bth	Counter ++
Bth < H/F	Counter = Max

(c)

Figure 29: Operation of the dynamic sharing scheme. (a) Excess caches have "monitoring sets" that track data flow-in and cache hit counts for each core. (b) The monitoring activity and excess cache allocation are done in accordance with a "monitoring period." When each monitoring period expires, the excess cache allocation to apply during the following monitoring period is determined. (c) Excess cache allocation counter calculation. It is done every excess cache allocation time. To provide large cache capacity for highly reused data quickly, the counter is set to the maximum value when high data reuse is detected.

having one monitoring set for every 32 sets works reasonably well.

To flexibly control excess cache allocation to individual cores, each core keeps an excess cache allocation counter. Figure 29(c) shows how these counters are set based on the ratio of flow-in and hit counts. We have four excess cache allocation actions: *decrease*, *no action*, *increase*, or *maximize*. When a burst access occurs from a core ($\frac{\text{hits}}{\text{flow-ins}} > B_{th}$), all excess caches are allocated to the core to quickly adapt to the demanding application phase. This is the maximize action. The number of allocated excess caches to a core is decreased when its hit count is zero ($\frac{\text{hits}}{\text{flow-ins}} = 0$). The rationale for this case is that if the core has many data flow-ins, but most data sweep through the excess caches without producing hits, the core should not use the excess caches. A core gets one more excess cache if it proves to benefit from excess caches ($M_{th} < \frac{\text{hits}}{\text{flow-ins}} < B_{th}$); otherwise ($\frac{\text{hits}}{\text{flow-ins}} < M_{th}$) the core will keep what it has. We heuristically determine 12.5% for B_{th}, 3% for M_{th} in our evaluation.

6.3 EVALUATION

6.3.1 Experimental setup

We evaluate StimulusCache with a detailed trace-driven CMP architecture simulator [18]. The parameters of the processor we model are given in Table 10. We select representative machine configurations to simulate: For an 8-core CMP, we simulated processor configura-

Table 10: Baseline CMP configuration.

Core's pipeline	Intel's ATOM-like two-issue in-order pipeline with 16 stages at 2GHz
Branch predictor	Hybrid branch predictor (4K-entry gshare, 4K-entry per-address w/ 4K-entry selector), 6 cycle mis-prediction penalty
Hardware prefetch	Four stream prefetchers per core, 16 cache block prefetch distance, 2 prefetch degree; implementation follows [69]
On-chip network	Crossbar for 8-core CMP and 2D mesh for 32-core CMP at half the core's clock frequency
On-chip caches	32KB L1 I-/D- caches with a 1-cycle latency; 512KB unified L2 cache with a 10-cycle latency; all caches use LRU replacement and have 64B block size
Memory latency	300 cycles

tions with 4 functional cores and 1, 2, 3, or 4 excess caches. For a 32-core CMP, we simulated processors with 16 functional cores and 4, 8, 12, or 16 excess caches.

We choose twelve benchmarks from SPEC CPU2006 [68], four benchmarks from SPLASH 2 [86], and SPECjbb 2005 [68]. Our benchmark selection from SPEC CPU2006 is based on working set size [27]; we picked a range of working set sizes to comprehensively evaluate the proposed policies under various scenarios. For workload preparation, we analyzed L2 cache accesses for the whole execution period of each benchmark with the reference input. Based on the analysis, we extracted each benchmark's representative excess cache interval, which includes the program's main functionality but skips its initialization phases. To evaluate a multiprogrammed workload, we use combinations of the single-threaded benchmarks. Tables 11(a) and (b) show the characteristics of the benchmarks selected and the multiprogrammed workloads. We simulate 10B cycles for single-threaded and multiprogrammed workloads. Other workloads (SPLASH-2 and SPECjbb 2005) are simulated to completion.

6.3.2 Results

6.3.2.1 Single-threaded applications The static private scheme is used for the single-threaded programs and all available excess cache is allocate to the program. Figure 30(a) shows the performance improvement of single-threaded applications with excess caches. Five

Table 11: Benchmark characteristics. (a) Benchmark selection. (b) Multiprogrammed workloads.

Suite	Characteristics (working set)	Benchmarks (working set size in 1B instructions)
SPEC CPU2006 INT	light moderate heavy	464.h264ref (5.5MB); 456.hmmer (2MB); 473.astar (26MB); 401.bzip2 (24.4MB); 429.mcf (680.8MB); 403.gcc (73MB);
SPEC CPU2006 FP	light moderate heavy	435.gromacs (8.6MB); 416.gamess (1.3MB); 450.soplex.2 (27.2MB); 483.sphinx3 (10.6MB); 459.GemsFDTD (800MB); 433.milc (230.8MB);
SPLASH-2	Multithreaded	fmm; ocean; lu; cholesky;
SPECjbb 2005	Server workload	100,000 transactions;

Combination	Applications
LLLL	464.h264ref, 456.hmmer, 435.gromacs, 416.gamess
LLMM1	464.h264ref, 435.gromacs, 473.astar, 483.sphinx3
LLMM2	464.h264ref, 435.gromacs, 401.bzip2, 450.soplex
LLMM3	456.hmmer, 416.gamess, 473.astar, 483.sphinx3
LLMM4	456.hmmer, 416.gamess, 401.bzip2, 450.soplex
LLHH1	464.h264ref, 435.gromacs, 429.mcf, 459.GemsFDTS
LLHH2	464.h264ref, 435.gromacs, 403.gcc, 433.milc
LLHH3	456.hmmer, 416.gamess, 429.mcf, 459.GemsFDTS
LLHH4	456.hmmer, 416.gamess, 403.gcc, 433.milc
MMMM	473.astar, 401.bzip2, 450.soplex. 483.sphinx3
MMHH1	473.astar, 483.sphinx3, 429.mcf, 459.GemsFDTS
MMHH2	473.astar, 483.sphinx3, 403.gcc, 433.milc
MMHH3	401.bzip2, 450.soplex, 429.mcf, 459.GemsFDTS
MMHH4	401.bzip2, 450.soplex, 403.gcc, 433.milc
HHHH	429.mcf, 403.gcc, 459.GemsFDTS, 433.milc

programs (`hmmer`, `h264ref`, `bzip2`, `astar`, and `soplex`) show more than 20% performance improvement while seven others had less improvement. Four heavy workloads (`gcc`, `mcf`, `milc`, and `GemsFDTD`) had almost no performance benefit from using excess caches. The different performance behavior can be interpreted from cache miss counts and cache miss reductions, shown in Figure 30(b) and (c), respectively. First, the four light workloads (`hmmer`, `h264ref`, `gamess`, and `gromacs`) have significant performance gains with excess caches because more cache capacity reduces a large portion of misses (42%–91%). However, their absolute miss counts are relatively small. In the case of `gamess`, the performance improvement was quite limited because it had almost no misses even without excess cache. Second, moderate integer workloads (`bzip2` and `astar`) have a pronounced benefit with excess cache due to their high absolute miss counts (4.4 and 11.9 per 1K instructions) and a good miss reduction of 44% and 55% each. Third, `soplex` sees a sizable performance gain with at least three excess caches. Figure 30(c) depicts the large miss reduction of `soplex` with four excess caches. It has a miss rate knee at around 2MB cache size (one local cache and three excess cache). Fourth, the heavy workloads (`gcc`, `mcf`, `milc`, and `GemsFDTD`) and one moderate workload (`sphinx3`) have little performance gain. The negligible miss reductions with excess cache explain this result. Our results clearly show that the static private scheme is in general very effective for improving individual program performance; there are sizable performance gains and no performance degradation. However, there are programs that do not benefit from excess caches at all.

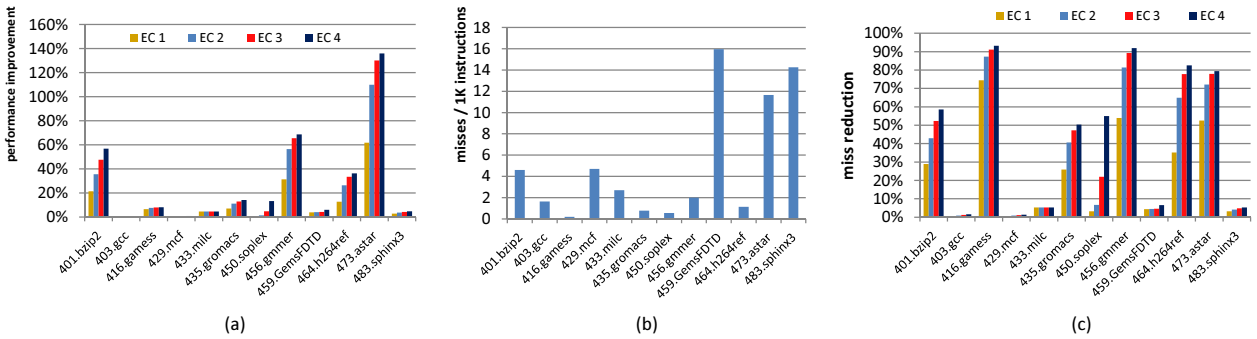


Figure 30: (a) Performance improvement of single-threaded applications with excess caches. (b) Misses per 1,000 instructions. (c) Miss reduction.

6.3.2.2 Multiprogrammed and multithreaded workloads

Static private scheme. Figure 31(a) shows the performance improvement of multiprogrammed and multithreaded workloads with the static private scheme. `LLMM3` had the largest improvement of 17%. The performance improvement of individual applications in the multiprogrammed workloads are depicted in Figure 31(b). When there is a large difference between the improvements of individual programs in a workload, the workload's overall performance improvement is limited by the application with the smallest individual gain. As shown in the previous subsection, there are programs that do not benefit at all from the use of excess caches.

For the multithreaded workloads, the static private scheme brought a large performance improvement for `lu` (45%) and `server` (42%). Other benchmarks had a 10% to 15% performance improvement. `lu` has a miss rate knee just after a total 512KB cache size. Therefore, adding one excess cache to each core has a great performance benefit. `server` has a high L2 cache miss rate of over 40% and lends itself to a large improvement given more cache capacity with excess caches. The multithreaded workloads we examined have symmetric behaviors (threads have similar cache demands) and all of them benefit from more cache capacity using the static private scheme.

Static sharing scheme. The multiprogrammed and multithreaded workloads can benefit from excess caches by sharing the extra capacity from the excess cache. Figure 32(a) shows

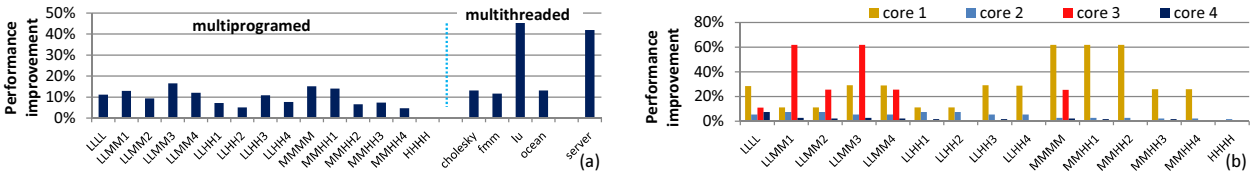


Figure 31: (a) Performance improvement with the static private scheme. (b) Performance improvement of individual programs.

performance improvement from employing a different number of excess caches with the static sharing scheme. The performance improvements of individual programs are shown in Figure 32(b). This figure presents the result when four excess caches were used.

For intuitive discussion, we categorize the multiprogrammed workloads into four groups. First, workloads in group 1 obtain significantly more benefits from the static sharing scheme than the static private scheme. They have at least two light programs and no heavy programs. Therefore, the programs in these workloads share excess cache capacity in a "fair" manner without thrashing. Second, workloads in group 2 exhibit limited relative performance benefit with the static sharing scheme compared to the static private scheme. In fact, the performance of `LLHH1` and `LLHH3` become worse with cache sharing. Performance degradation can be caused by the heavy programs that use up the entire excess cache capacity, sacrificing the performance improvement opportunities of co-scheduled, light programs. Third, workloads in group 3 show sizable performance gains from cache sharing because `astar` greatly benefits from more cache capacity. Figure 32(b) shows that `astar` has 135% performance improvement regardless of other co-scheduled programs. Fourth, workloads in group 4 have very small performance improvement from excess cache sharing. Clearly, simply sharing cache capacity without considering the program mix does not result in a performance improvement.

Multithreaded and server workloads have nearly identical performance improvement as the static private scheme. This result suggests that these workloads can readily exploit the given excess cache capacity with the simple static private and static sharing schemes because

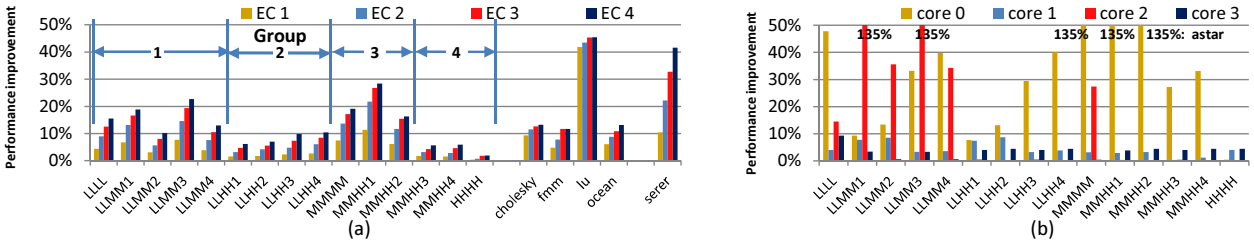


Figure 32: (a) Performance improvement with the static sharing scheme. Workloads are grouped into: Group 1: "Large gain," Group 2: "Limited gain due to heavy applications," Group 3: "Large gain due to `astar`," and Group 4: "Small gain." (b) Performance improvement of individual programs with four excess caches. `astar` consistently shows a high gain of 135%.

the threads have balanced cache capacity demands.

Dynamic sharing scheme. The dynamic sharing scheme has the potential to overcome the deficiency of the static sharing scheme, which does not avoid destructive competition in some co-scheduled programs. Figure 33(a) shows the overall performance gain using the dynamic sharing scheme. As the dynamic sharing scheme is suited to situations when co-scheduled programs aggressively compete with one another, our presentation focuses on the multiprogrammed workloads.

The benefit of dynamic sharing is significant when there are heavy programs, especially for group 2 workloads in Figure 33(a) and Figure 32(a). Moreover, the relative benefit is pronounced with a smaller number of excess caches. For example, workloads with a variety of memory demands (e.g., `LLHH1`–`LLHH4` and `MMHH3`–`MMHH4`) gain large benefits from the dynamic sharing scheme with only one excess cache. Figure 33(b) presents the relative benefit of the dynamic sharing scheme to the static sharing scheme when four excess caches are given. The result shows that group 1 workloads have little additional performance gain because the static sharing scheme already achieves high performance in the absence of cache trashing programs. However, `LLMM2` and `LLMM4` still show measurable additional performance gain with the dynamic sharing scheme. Second, group 2 workloads have the highest additional performance improvement with dynamic sharing. All four workloads have at least one program which achieves additional performance improvement of 5% or

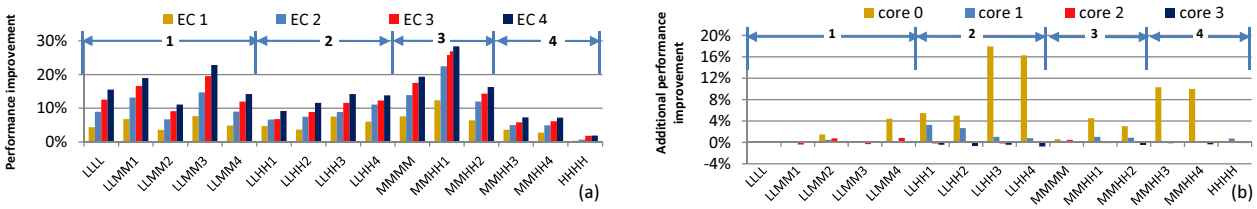


Figure 33: (a) Performance improvement with the dynamic sharing scheme. (b) Additional performance gain with the dynamic sharing scheme compared with the static sharing scheme with four excess caches. Workloads are grouped into: Group 1: "Limited gain," Group 2: "Large gain," Group 3: "Limited gain," and Group 4: "Large gain of two programs."

more (5.5%, 5.1%, 16.8%, and 16.2%). On the other hand, some programs actually suffer performance degradation because the dynamic sharing scheme strictly limited their use of excess cache capacity. However, the performance degradation is very limited—the largest performance degradation observed was only 0.6% (`milc` of `LLHH4`). Third, group 3 workloads show only a small additional performance gain as the large performance potential of adding more cache capacity has been already achieved with the static sharing scheme. However, there were noticeable additional gains for `MMHH1` and `MMHH2` which have heavy programs. Fourth, `bzip2` in group 4 has a large additional performance gain with the dynamic sharing scheme. The other programs in this group get negligible benefit.

The results demonstrate the capability of the proposed dynamic sharing scheme in StimulusCache; it can robustly improve throughput of multiprogrammed workloads without unduly penalizing individual programs. The dynamic and adaptive control of excess cache resources allocation among competing co-scheduled programs is shown to be critical to get the most from the available excess caches.

6.3.2.3 Comparing StimulusCache with Dynamic Spill-Receive (DSR) To put StimulusCache in perspective, we compare it with a well known dynamic spill-receive (DSR) scheme [71] which effectively utilizes multiple private caches among co-scheduled programs. Cooperative caching (CC) [16] and DSR are two representative private L2 cache schemes, which could be used to merge excess caches. We chose to compare StimulusCache with DSR

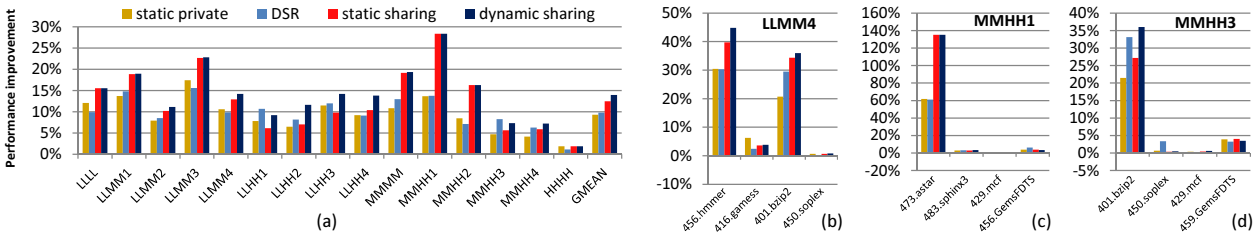


Figure 34: (a) Performance improvement with three StimulusCache policies and Dynamic Spill-Receive (DSR). (b)–(d) Performance improvement of individual programs in three example workloads: (b) `LLMM4` (DSR < static private < static sharing < dynamic sharing); (c) `MMHH1` (static private = DSR << static sharing = dynamic sharing); and (d) `MMHH3` (static private < static sharing < dynamic sharing < DSR).

because it has better performance than CC for many workloads [71].

Figure 34(a) presents the performance improvement with StimulusCache's three policies and DSR, given four excess caches. Overall, StimulusCache's dynamic sharing and static sharing schemes achieve substantially better performance than DSR. DSR shows the least performance improvement for quite a few workloads (`LLLL`, `LLMM3`, `LLMM4`, `LLHH4`, `MMHH2`, and `HHHH`). Only two workloads (`LLHH1` and `MMHH3`) have better performance improvement with DSR. Figure 34(b)–(d) show individual performance improvement in selected workloads. It is shown that programs like `hmmer`, `bzip2` (in `LLMM4`) and `astar` perform significantly better with StimulusCache than DSR. On the other hand, `soplex` in `MMHH3` performed better with DSR. Even in this workload, the three other programs in `MMHH3` perform better with StimulusCache.

DSR's relatively poor performance comes partly from the fact that it does not differentiate excess caches from other local L2 caches. Excess caches are strictly remote caches and are not directly associated with a particular core. Hence, an excess cache should be a "receiver" in the context of DSR. However, DSR's spiller-receiver assignment decision for each cache is skewed as there are no local cache hits or misses for the excess caches, and surprisingly, the excess caches become a "spiller" from time to time, which blocks their effective use as additional cache capacity. Furthermore, unlike the excess cache chain of the dynamic sharing scheme, a miss in one-level receiver caches in DSR is a global miss. DSR

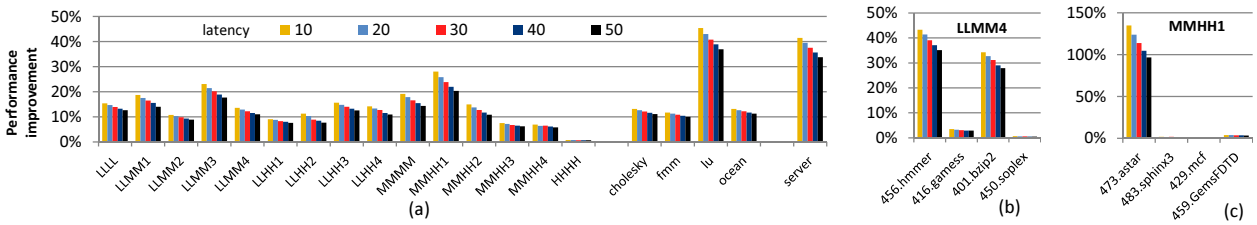


Figure 35: (a) Performance improvement when excess cache latency is varied. (b)–(c) Individual program's performance improvement for (b) `LLMM4` and (c) `MMHH1`.

provides a much shallower LRU depth than StimulusCache. Therefore, even if we designate excess caches as receivers in DSR, it does not perform as well as the dynamic sharing scheme of StimulusCache.

6.3.2.4 Network traffic Excess caches may introduce additional network traffic due to staggered cache access to multiple excess caches and downward block propagation. A single local cache miss can cause N data propagations from the local cache to the main memory with N excess caches. An excess cache hit generates K block propagations if the K'th excess cache had a hit. Our experiments revealed that StimulusCache does not increase the network traffic significantly. The average on-chip network bandwidth usage per core was measured to be 155.1MB/s (`cholesky`) to 517.3MB/s (`MMHH1`) without excess cache. With excess cache, the bandwidth usage was 187.5MB/s (`fmm`) to 873.7MB/s (`MMHH1`), well below the provided network bandwidth capacity of 8GB/s per core. The increase was 101.7MB/s on average and up to 423.2MB/s (`LLMM3`). Note that the reduced execution times with StimulusCache also push up the network bandwidth usage.

6.3.2.5 Excess cache latency In this section, we study the sensitivity of program performance to excess cache access latency. Long excess cache latencies may result from slower on-chip networks, network contention, or non-uniform distances between the program location and the excess cache locations. Figure 35 shows the performance improvement of various workloads with excess caches having varied latencies. While the performance improvement

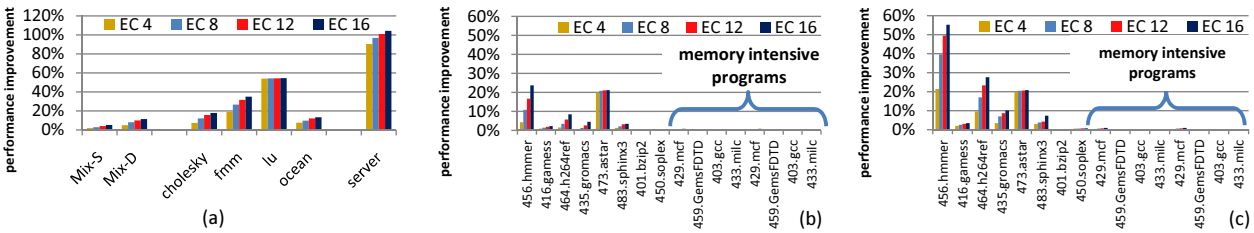


Figure 36: Performance improvement of 32-core CMPs with excess caches. (a) Overall throughput improvement. (b)–(c) Performance improvement of individual programs (b) with the static sharing scheme, and (c) with the dynamic sharing scheme.

decreases with an increase in latency, the overall performance improvement remains significant, even with the longest latency of 50 cycles.

The performance impact due to long excess cache latencies is limited because accesses hit more frequently in the local L2 cache and in the first few excess caches. The extent of the impact varies from workload to workload depending on how frequently an access has to travel further down the excess cache chain. Figure 35(b) and (c) further show that the performance impact varies from program to program within a single multiprogrammed workload. For example, in `LLMM4`, `hmmer` and `bzip2` were measurably affected by the increased excess cache latency. The other two programs in the workload were not. This result is intuitive because the programs that get more benefit from the excess caches could be impacted more from the increased latency.

6.3.2.6 32-core CMP Finally, we experimented with a 32-core CMP configuration, where 16 cores run programs and there are 4–16 excess caches. We use the static sharing scheme for multithreaded and server workloads and the dynamic sharing scheme for multiprogrammed workloads. We use a multiprogrammed workload that encompasses all twelve SPEC2006 benchmarks listed in Table 11(a). Additionally, a second copy of the four heavy workloads are run on cores 13 to 16 to ensure that all 16 functional cores are kept busy. Figure 36(a) shows the overall performance improvement with excess cache.

We make two observations. First, the dynamic sharing scheme for the multiprogrammed

workload works well for this large-scale CMP. Using 16 excess caches, the dynamic sharing scheme yields a performance improvement of 11% whereas the static sharing scheme's improvement is only 5%. The superiority of the dynamic sharing scheme is more clearly revealed in Figure 36(b) and (c). Second, the multithreaded and server workloads also have large performance improvements. `lu` and `server` have 54.2% and 90.5% improvement with only four excess caches, respectively. The performance improvement of `server` is as high as 104.4% with 16 excess caches. This result underscores the importance of on-chip memory optimization for memory-intensive workloads with large footprints.

6.4 SUMMARY

CMPs are expected to have many processor cores and cache resources. Given higher integration and smaller device sizes, maintaining chip yield above a profitable level remains a challenge. As a result, various on-chip resource isolation strategies will become important. Traditional core-disabling techniques to tackle low yield abandon large on-chip cache when a corresponding core is unusable.

This chapter proposes StimulusCache, where we decouple private L2 caches from their cores and salvage unemployed L2 caches when the corresponding cores become unavailable due to hardware faults. We explore how available excess caches can be used and develop effective excess cache utilization policies. For single-threaded programs, StimulusCache offers a sizable benefit by reducing up to 91% of L2 misses and increasing program performance by up to 131%. Our unique logical chaining of excess caches exposes an opportunity to control the usage of the shared excess caches among multiple co-scheduled programs. In most cases we examine, programs secure at least 90% of the benefit they would get when monopolizing all available excess caches. Using this technique, multiple applications gain 90% of the benefit from the single application running. We also show that the additional hardware design cost is small.

7.0 CONCLUSIONS

7.1 SUMMARY

This thesis proposes and studies fault- and yield-aware on-chip memory design and management. Traditionally, processor architects focused on performance, area, and power/energy aspect. Guaranteeing fault tolerance and enhancing chip yield have been largely a separate effort made by low-level circuit quality engineers, layout designers, and process engineers. However, as chips built with deep sub-micron technology are more susceptible to manufacturing defects, process variations, and aging phenomena, it becomes imperative to consider reliability and yield together at an early design time by the processor architect/cache designer.

As a first step, we propose DEFCAM, a novel cache design framework that integrates cache defect, yield, and performance models, and lays a solid foundation for evaluating a cache memory designed on nanometer-scale technology in terms of its yield, area, and performance. A new metric called YAP (yield-area-performance) is introduced, which enables a designer to quickly evaluate different cache designs with a single number. Depending on the design goals, customized YAPs (e.g., $Y^2A^{-.7}P$) can be defined for different design optimization interests. Using DEFCAM and the YAP metric, a cache designer can directly compare cache designs that have different architectural, organizational, and defect-related parameters at an early design stage.

With DEFCAM, we evaluated the performance impact from three architecturally visible fault classes (line, set, and way). Among three fault classes, we discovered that only set faults have critical performance impact because they create non-cacheable address holes. To tackle set faults, set remapping is proposed, which redirects memory accesses to faulty sets to

available sound cache sets. This thesis also shows that set remapping can be done statically or dynamically with off-line or on-line profile information. We found that static remapping is the simplest in design complexity. It also achieves much of the potential of an oracle based approach. Thus, we recommend static remapping as a cost effective fault masking technique for L1/L2 caches.

To illustrate how DEFCAM can be used in practice, we performed a case study to compare a number of cache yield management strategies, including: redundant row, ECC, line delete, and set remapping. When the redundant row and ECC schemes are used in isolation, they fall short of other degradation-based schemes quickly as the effect of defect and process variations is increased. Because these schemes do not allow line disabling with degraded performance, yield is drastically decreased with high error rates. If more redundant rows are added to come up with the high error rates, overall area overhead becomes too large. Degradation-based schemes such as line delete and set remapping were shown to offer much higher yield with limited area overheads. It turned out that a little lower performance target (i.e., 99%) greatly improve the chip yield. Among the cache yield management strategies, set remapping consistently offers the highest YAP metrics under various defect and process variation scenarios, which can be utilized in L1/L2 cache design.

For on-chip directory memory, this thesis proposed PERFECTORY, a comprehensive, low-overhead microarchitectural scheme to detect and correct hardware errors that occur in the coherence directory memory. PERFECTORY utilizes coherence protocol characteristics of the exclusive and shared states in the MESI protocol (modified state is regarded as exclusive state). For exclusive state, multiple pointer-ECC pairs are used to utilize the inherent redundancy of the sharer bits in the state. For shared state, we developed a read-time detection scheme which finds hard fault bits by writing the original data and the XORed data. The XNOR value of the two values should be zero if there is no fault bit. Once faults are found, invalidation messages are always sent to the cores with faulty bits. For soft-error, one parity bit is used for sharer block which detects one bit flip in addition to any number of hard faults on the block.

Evaluation results show that PERFECTORY achieves very high chip yield–at 0.2% HER, yield is 100% with PERFECTORY, whereas a 32-way row redandancy scheme's yield is

virtually zero. Furthermore, PERFECTORY significantly improves life-time reliability of the directory memory. The MTTF using a 100-processor cluster configuration is 1,934 years with PERFECTORY at 0.05% HER and 1,000 FIT, whereas ECC's MTTF is only 36 days under the same condition. Compared with existing yield improvement strategies and soft error protection mechanisms, PERFECTORY has a substantially smaller chip area cost and performance slowdown. It requires only one bit per directory entry. Finally, PERFECTORY shows less than 0.1% program performance degradation at 0.05% HER whereas a state-of-the-art disabling scheme has 4% program performance degradation.

Lastly, this thesis proposes StimulusCache where we decouple private L2 caches from their cores and salvage unemployed L2 caches when the corresponding cores become unavailable due to hardware faults. To support excess caches, we designed a new data structure in the cache controller, which includes excess cache allocation vector, shared core vector, and next excess cache pointers. With these vector tables, each sound core can flexibly access any L2 cache bank in unused cores. Three utilization policies were proposed: static private, static sharing, and dynamic sharing. Static private provides performance isolation but shows limited capacity utilization because only one core exclusive access the excess cache. Static sharing expands the limit of static private by allowing multiple cores to access excess caches. However, it is vulnerable for the capacity threshing from a few memory hungry applications. To avoid capacity threshing, dynamic sharing monitors the capacity benefits for each thread, and allocates extra excess caches only to beneficial cores.

Experimental results show that StimulusCache significantly improves overall throughput. For single-threaded programs, StimulusCache offers a sizable benefit by reducing up to 91% of L2 misses and increasing program performance by up to 131%. Our unique logical chaining of excess caches exposes an opportunity to control the usage of the shared excess caches among multiple co-scheduled programs. In most cases we examine, programs secure at least 90% of the benefit they would get when monopolizing all available excess caches. Using this technique, multiple applications gain 90% of the benefit from the single application running. For multi-threaded programs, static sharing shows 13%-45% performance improvement on SPLASH2 benchmarks. Dynamic sharing is very efficient for multi-programmed workloads where multiple threads are competing each other for capacity utilization. We also show that

additional hardware design cost is small.

7.2 CONCLUSIONS

This work proposes architectural approaches toward fault- and yield-aware on-chip memory design and management. First, we built a design and evaluation framework which enabled architects to quickly compare various on-chip memory designs for various faults environments including physical defects and process variations. Based on the framework, architectural memory design techniques were proposed for yield improvement of various on-chip memory elements including L1/L2 cache and directory memory. The following conclusions can be drawn from the qualitative analysis and the experimental results.

- Faults have different impacts on yield and performance depending on the fault location in a chip. Set fault has the most crucial impact on performance among line, set, and way faults. Based on this observation, set remapping is proposed, which greatly outperforms previous fault masking techniques, such as redundancy, ECC, and grace degradation in terms of YAP.

- There is a wide yield discrepancy between compute cores and L2 caches because many yield improving techniques are available to improve yield of regular structured L2 cache memory cell array. This causes to have many unused excess L2 caches in disabled core with core disabling technique. Utilizing these excess caches to help other cores' performance has great benefit for a wide range of workloads.

- Cache coherence protocol is used to design efficient fault masking techniques for on-chip directory, which utilize inherent redundancy of exclusive state's sharer encoding and false invalidation for shared state. These architectural approaches have good yield and reliability improvement with negligible performance overhead.

7.3 FUTURE WORK

This thesis opens a new area on a yield-aware on-chip memory design methodology research. The study demonstrates that architectural yield improvement techniques are promising approaches. There are sill many interesting research topics that can be considered as the possible future work:

- The evaluation framework presented in this thesis can only analyze 6-T SRAM cell array based memories. As processor design is evolving to include many other on-chip memory elements such as eDRAM, the framework can be further improved to include futuristic on-chip memory elements.
- There are many memory resources beyond the core boundary to be investigated to improve chip yield, including on-chip network and memory controllers. Given that the core count and memory bandwidth requirements will increase, the chip area for on-chip network and memory controllers will also grow significantly. To improve the overall chip yield without compromising performance, efficient fault isolation should be investigated. Similar to the thesis approaches, micro-scopic and macro-scopic techniques should be developed.
- This thesis assumed in-order (IO) processor model to evaluate expected performance of CMPs. However, many future CMPs may adopt out-of-order (OoO) cores as the transistor budget increases. Thus, integrating OoO core performance model to the design framework would greatly expand the usability of the framework. Once OoO core model is embedded, the framework would be applicable to design yield aware on-chip memory elements in heterogeneous CMPs which have OoO and IO cores.

BIBLIOGRAPHY

[1] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz. "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of International Symposium on Computer Architecture (ISCA)*, May 1988.

[2] A. Agarwal, B. C. Paul, H. Mahmoodi-Meimand, A. Datta, and K. Roy. "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies," *IEEE Transaction on VLSI System (TVLSI)*, Jan. 2005.

[3] D. H. Albonesi. "Selective cache ways: On-demand cache resource allocation," *Proceedings of International Symposium on Microarchitecture (MICRO)*, pp. 248–259, 1999.

[4] AMD. "BIOS and Kernel Developer's Guide for AMD Athlon64 and AMD Opteron Processors," http://support.amd.com/us/Processor_TechDocs/26094.pdf.

[5] AMD Phenom Processors. http://www.amd.com,

[6] E. A. Amerasekera and F. N. Nasm. "Failure mechanisms in semiconductor devices," Jan. 1997.

[7] H. Ando, K. Seki, S. Sakashita, M. Aihara, Kan, and K. Imada. "Accelerated Testing of a 90nm SPARC64 V Microprocessor for Neutron SER," *IEEE Workshop on silicon Errors in Logic - System Effects (SELSE)*, Jan. 2007.

[8] C. Argyrides, H. R. Zarandi, and D. K. Pradhan. "Matrix Codes: Multiple Bit Upsets Tolerant Method for SRAM Memories," *IEEE International Symposium on Defect and Fault-Tolerance in VLSI System (DFT)*, pp. 26–28, Sep. 2007.

[9] T. Austin, E. Larson, and D. Ernst. "Simplescalar: An infrastructure for computer system modeling," *IEEE Computer*, 35, 2, pp. 59–67, 2002.

[10] Berkeley Predictive Technology Model. http://www-device.eecs.berkeley.edu/ ptm/.

[11] S. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, J. R. Rattner. "Platform 2015: Intel Processor and Platform Evolution for the Next Decade," *Technology@Intel Magazine*, Mar. 2005.

[12] S. Borkar. "Microarchitecture and Design Challenges for Gigascale Integration," *keynote speech at International Symposium on Microarchitecture (MICRO)*, Dec. 2004.

[13] D. K. Bhavsar. "An Algorithm for Row-Column Self-Repair of RAMs and Its Implementation in the Alpha 21264," *Proceedings of International Test Conference (ITC)*, Sep. 1999.

[14] D. C. Bossen, J. M. Tendler, and K. Reick. "Power4 System Design for High Reliability," *IEEE Micro*, Jan. 2002.

[15] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. "Tolerating Hard Faults in Microprocessor Array Structure," *Proceedings of Dependable Systems and Networks (DSN)*, Jun. 2004.

[16] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2006.

[17] J. Chang, M. Huang, J. Shoemaker, J. Benoit, S. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, S. Rusu, and D. Srivastava. "The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series," *IEEE Journal of Solid-State Circuits (JSSC)*, Apr. 2007.

[18] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng. "TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation," *Proc. Int'l Conf. Parallel Processing (ICPP)*, pp. 446–453, Sep. 2008.

[19] S. Cho and L. Jin. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," *Proceedings of International Symposium of Microarchitecture(MICRO)*, pp. 455–465, Dec. 2006.

[20] D. E. Culler and J. P. Singh. "Parallel Computer Architecture A HW/SW Approach," Morgan Kaufmann Publishers 1998.

[21] J. Dorsey, . "A Fault-Driven, Comprehensive Redundancy Algorithm," *IEEE Design & Test of Computers (DT)*, Jun. 1985.

[22] S. M. Domer, S. A. Foertsch, and G. D. Raskin. "Component Level Yield/Cost Model for Predicting VLSI Manufacturability on Designs Using Mixed Technologies, Circuitry, and Redundancy," *IEEE Custom Integrated Circuits Conference (CICC)*, pp. 313–316, May 1994.

[23] S. M. Domer, S. A. Foertsch, and G. D. Raskin. "Model for Yield and Manufacturing Prediction on VLSI Designs for Advanced Technologies, Mixed Circuitry, and Memory," *IEEE Journal of Solid-State Circuits (JSSC)*, Mar. 1995.

[24] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. "An Integrated Quad-Core Opteron Processor," *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2007.

[25] D. J. Friedman, M. H. Hanse, V. N. Nair, and D. A. James. "Model-Free Estimation of Defect Clustering in Integrated Circuit Fabrication," *IEEE Transactions on Semiconductor Manufacturing (TOSM)*, pp. 344–359, Aug. 1997.

[26] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. "Architecture and Design of AlphaServer GS320," *Proceedings of International Conference of Architectural Support for Programming Languages and Operating systems (ASPLOS)*, Dec. 2000.

[27] D. Gove. "CPU2006 Working Set Size," *ACM SUGARS Computer Architecture News*, 35(1):90–96, Mar. 2007.

[28] M. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "Mibench: A free, commercially representative embedded benchmark suite," *Proceedings of Workshop Workload Characterization (WWC)*, Dec. 2001.

[29] R. Halfhill. "Ambric's New Parallel Processor," *Microprocessor Report (MPR)*, Oct. 2006.

[30] D. A. Hodges. *Analysis and Design of Digital Integrated Circuits in Deep Submicron Tech.*, 3rd Ed., McGraw-Hill, 2003.

[31] M. Y. Hsiao. "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, 14(4):395–401, Jun. 1970.

[32] IBM. "IBM System p570 with new POWER6 processor increases bandwidth and capacity," *IBM US Hardware announcement*, pp. 107–288, May 2007.

[33] Intel ATOM Processors. http://www.intel.com/technology/atom/,

[34] Intel Corp. "Intel Microarchitecture, Codenamed Nehalem," technology brief, http://www.intel.com/technology/architecture-silicon/next-gen/,

[35] ITRS. "International technology roadmap for semiconductors," http://public.itrs.net, 2007.

[36] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. "Distributed-Directory Scheme: Scalable Coherent Interface," *IEEE Computer (HPCRI)*, pp. 74–77, Jun. 1990.

[37] R. Joseph. "Exploring Salvage Techniques for Multi-core Architectures," *In Workshop High Performance Computing Reliability Issues (HPCRI)*, Feb. 2005.

[38] R. Kalla. "POWER7: IBM's Next Generation POWER Microprocessor," *Presentation at A Sumposium on High Performance Chips (HOTCHIPS)*, Aug. 2009.

[39] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. "Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding," *Proceedings of International Symposium on Microarchitecture (MICRO)*, Dec. 2007.

[40] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, Mar./Apr. 2005.

[41] S. Kumar, C. Kim, and S. Sapatnekar. "Impact of nbti on sram read stability and design for reliability," *Proceedings of International Symposium on Quality Electronics Design(ISQED)*, Mar. 2006.

[42] D. Lamet, J. F. Frenzel. "Defect-Tolerant Cache Memory Design," *Proceedings of IEEE VLSI Test Symposium (VTS)*, Apr. 1993.

[43] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server," *Proceedings of International symposium of Computer Architecture (ISCA)*, June 1997.

[44] H. Lee, S. Cho, and B. R. Childers. "StimulusCache: Boosting Performance of Chip Multiprocessors with Excess Cache," *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2010.

[45] H. Lee, S. Cho, and B. R. Childers. "DEFCAM: A Design and Evaluation Framework for Defect-Tolerant Cache Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, submitted. in major revision.

[46] H. Lee, S. Cho, and B. R. Childers. "PERFECTORY: A Fault-Tolerant Directory Memory Architecture," *IEEE Transactions on Computers (TC)*, accepted.

[47] H. Lee, S. Cho, and B. R. Childers. "Performance of graceful degradation for cache faults," *IEEE Computer Society Annual Symposium on VLSI(ISVLSI)*, pp. 409–415, May 2007.

[48] H. Lee, S. Cho, and B. R. Childers. "Exploring the Interplay of Yield, Area, and Performance in Processor Caches," *Proceedings of International Conference on Computer Design (ICCD)*, Oct. 2007.

[49] A. S. Leon, B. Langley, J. S. Jinuk. "The UltraSPARC T1 Processor: CMP Reliability," *Proceedings IEEE Custon Integrated Circuits Conference (CICC)*, Mar. 2006.

[50] M. R. Marty and M. D. Hill. "Virtual Hierarchies to Support Server Consolidation," *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2007.

[51] P. Mazumder. "An On-Chip Double-Bit Error-Correcting Code for Three-Dimensional Dynamic Random-Access Memory," *IEE international Test Conference (ITC)*, pp. 279–288, Sept. 1988.

[52] A. Meixner and D. J. Sorin. "Detouring: Translating Software to Circumnevt Hard Faults in Simple Cores," *Proceedings of Dependable Systems and Networks (DSN)*, Jun. 2008.

[53] S. Naffziger, B. Stackhouse, and T. Grutkowski. "The Implementation of A 2-Core Multi-Threaded Itanium-Family Processor," *Proceedings IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2005.

[54] NVIDIA Tesla GPU. http://www.nvidia.com/object/tesla_computing_solutions.html,

[55] NVIDIA GeForce 8800 GPU. http://www.nvidia.com,

[56] Y. Ooi, M. Kashimura, H. Takeuchi, E. Kawamura. "Fault-Tolerant Architecture in a Cache Memory Control LSI," *IEEE Journal of Solid-State Circuits (JSSC)*, Apr. 1992.

[57] D. A. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. Van Dyke. "Architecture of a vlsi instruction cache for a risc," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 108–115, Jun. 1983.

[58] M. Plakal, D. H. Sorin, A. E. Condon, M. D. Hill. "Lamport clocks: verifying a directory cache-coherence prorocol," *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, Jun.–Jul. 1998.

[59] D. W. Plass and Y. H. Chan. "IBM POWER6 SRAM arrays," *IBM Journal of Research and Development*, 51(6):747–756, Nov. 2007.

[60] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. "Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance," *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2009.

[61] A. F. Pour and M. D. Hill. "Performance implications of tolerating cache faults," *IEEE Transactions on Computers(TC)*, pp. 257–267, Mar. 1993.

[62] G. S. Sohi. "Cache Memory Organization to Enhance the Yield of High-Performance VLSI Processors," *IEEE Transactions on Computers (TC)*, Apr. 1989.

[63] SEMATECH. "Critical Reliability Challenges for ITRS," *Tech Transfer #03024377A-TR*, Mar. 2003.

[64] E. Sperling. "Thrn Down the Heat ... Please—Interview with Tom Reeves of IBM," *EDN*, Jul. 2006.

[65] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 system microarchitecture," *IBM Journal of Research & Development*, Jul./Sep. 2005.

[66] C. W. Slayman. "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Trans. on Device and Materials Reliability*, 5(3), Sept. 2005.

[67] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. "The impact of technology scaling on lifetime reliability," *Proceedings of International Conference on Dependable Systems and Networks(DSN)*, Jun. 2004.

[68] Standard Performance Evaluation Corporation. http://www.specbench.org.

[69] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. "POWER4 system microarchitecture," *IBM Techical White Paper*, Oct. 2001.

[70] N. Quach. "High availability and reliability in the Itanium processor," *IEEE Micro*, 20(5):61–69, Sep.-Oct. 2000.

[71] M. K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2009.

[72] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. "Adaptive Insertion Policies for High-Performance Caching," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 381–391, Jun. 2007.

[73] M. K. Qureshi and Y. N. Patt. "Utility-Based Partitioning of Shared Caches," *Proceedings of International Symposium on Microarchitecture (MICRO)*, Dec. 2006.

[74] S. Rusu, S. Tam, H. Mulijono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Korrapalli, "A 45nm 8-core enterprise xeon processor," *IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2009.

[75] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. "VARIUS: A Model of Process Variation and Resulting Timing Errors for Microarchitects," In *IEEE Transactions on Semiconductor Manufacturing (TSM)*, 21(1): 3–13, Feb. 2008.

[76] M. Spica and T. Mak. "Do we need anything more than single bit error correction ECC?," In *International Workshops on Memory Technology, Design and Testing*, 2004.

[77] C. H. Stapper and H. -S. Lee. "Synergistic fault-tolerance for memory chips," In *IEEE Transactions on Computers (TC)*, Sep. 1992.

[78] P. Sweazey and A. J. Smith. "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *Proceedings of International Symposium on Computer Architecture(ISCA)*, May 1986.

[79] T. Tang, V. De, and J. D. Meindl. "Intrinsic mosfet parameter fluctuations due to random dopant placement," *IEEE Transactions on VLSI Systems*, Dec. 1997.

[80] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. "Cacti 4.0," *HP Lab. Tech. Rep. HPL-2006-86*,

[81] T. Thomas and B. Anthony. "Area, performance, and yield implications of redundancy in on-chip caches," *Proceedings of International Conference on Computer Design (ICCD)*, pp. 291–292, Feb. 1999.

[82] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. "Cacti 5.1 Technical report," HP Laboratories, Palo Alto, 2008.

[83] T. Wada, S. Rajan, and S. A. Przybylski. "An analytical access time model for on-chip cache memories," *IEEE Journal of Solid-State Circuits (JSSC)*, pp. 1147–1156, Aug. 1992.

[84] C. Webb. "45nm Design for Manufacturing," Intel Technology Journal, Nov. 2008.

[85] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S. Lu. "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation," *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2008.

[86] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 24–36, July 1995.

[87] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. H. Irwin. "The design and use of simplepower: A cycle-accurate energy estimation tool," *Proceedings of Design Automation Conference (DAC)*, Jun. 2000.

[88] C. Zhang. "Balanced cache: Recuding conflict misses of direct-mapped caches through programmable decoders," *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2006.

[89] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2005.